Application of Convolutional Neural Networks to Classify Crucible Position Inside an Analytical Machine

Bachelor Thesis of Carine Silva Allen

Matriculation-No.: 24404

Mechanical Engineering B.Sc.

Faculty of Technology and Bionics

Rhine-Waal University of Applied Sciences

Supervised by:
Prof. Dr.-Ing. Dirk Nissing
Dr. Rainer Schramm

Winter Semester 2021/22

Declaration of Authenticity

I, Carine Silva Allen, hereby declare that the work presented here is my own work completed without the use of any aids other than those listed. Any material from other sources or works done by others has been given due acknowledgement and listed in the reference section. The work presented here has not been published or submitted elsewhere for assessment in the same or a similar form.

_____ Kleve, 17 January 2022

Carine Silva Allen

ACKKNOWLEDGEMENTS

Praise the Lord, God Almighty, for your grace is sufficient. I express my heartfelt gratitude to you, my Lord, for without your blessings, none of this would have been possible.

I would like to show my greatest appreciation to Prof. Dr.-Ing. Dirk Nissing as the thesis supervisor, for his unwavering support, guide and help in the process of making this thesis. Thank you for being patient and always taking the time to answer all my questions

I want to express my gratitude to Dr. Rainer Schramm for his constructive remarks and suggestions for improving this study, as well as for being the study's founder and sponsor. This thesis would not be possible without the set of knowledge that I acquired during my years working at FLUXANA.

Finally, I would like to thank my mom Lucyjane and my sister Caroline for their continuous support and encouragement throughout the academic year and throughout the completion of this thesis and no words can express how grateful I am for all of your sacrifices and devotion. Thank you very much!

ABSTRACT

Workplace technological developments go beyond assuring worker productivity, fostering creativity, and boosting company margins. It helps promote and improve the wellness of both employees and employers by minimizing the number of accidents. In this thesis, an image analysis method is used to help prevent that an accident occurs when a wrongly positioned crucible enters the furnace for fusion process inside of an analytical machine.

For that, machine learning models are used. Convolutional Neural Networks are special type of Neural Networks that are well suited for image processing and have had tremendous success in recent years on a variety of machine learning challenges. This thesis is focused on the configuration and performance comparison of three different Convolutional Neural Network architectures on a crucible position classification task. The performance of these three Convolutional Neural Network architectures is compared not only by their accuracy, but also on their robustness on position classification in different light scenarios. These three architectures are: SSD with MobileNetV2; Faster R-CNN with ResNet-101 and Faster R-CNN with Inception ResNet V2.

The Results show that the Faster R-CNN with Inception ResNet V2 model is the most suitable when classifying only the handle position of a crucible. It shows a 100% of correct position classifications on all exposure times. When classifying only the axis position of a crucible, the Faster R-CNN with ResNet-101 show the better results, with an accuracy above 90% to all exposure times while being the most stable model when changing the light conditions, with a variance of less than 10% in the position classifications when light goes a little darker or brighter.

Keywords: Machine Learning, Convolution Neural Networks, Position Classification, SSD, MobileNetV2, Faster R-CNN, ResNet-101, Inception ResNet V2.

TABLE OF CONTENTS

LIST OF ABBREVIATIONS

Abbreviation	Definition
ACC	Accuracy
AI	Artificial Intelligence
ANN	Artificial Neural Network
API	Application Programming Interface
BN	Batch Normalization
CNN	Convolutional Neural Network
CPU	Central Processing Unit
ERR	Error
FC	Fully Connected layer
GPU	Graphical Processing Unit
GUI	Graphic User Interface
mAP	mean Average Precision
R-CNN	Region-based Convolutional Neural Network
ReLU	Rectified Linear Unit
ResNet	Residual Network
R-FCN	Region-based Fully Convolutional Neural Network
RoI	Region of Interest
SSD	Single Shot Multibox Detection
TPU	Tensor Processing Unit
XRF	X-ray Fluorescence

LIST OF FIGURES

LIST OF TABLES

1 INTRODUCTION

The purpose of this thesis is to apply different Convolutional Neural Networks (CNNs) models into a crucible position classification task, so that the performance can be evaluated and compared. In this first chapter, the motivation and goal of the thesis are presented, as well as the thesis structure.

1.1 MOTIVATION

FLUXANA® it's specialized in creating application solutions for X-ray fluorescence (XRF). The inorganic components of a substance/product can be determined using this spectroscopic method, which is utilized in elemental analysis. The strategy is employed in the industry in quality assurance as well as in institutions and government organizations that keep track of regulatory compliance.

A variety of equipment for sample preparation is fabricated at FLUXANA®, among them, the fusion machine VITRIOX® ELECTRIC that is a fully automatic fusion machine with cold to cold operation for 1, 2, 4 or 6 samples (1 sample at a time) [1]. The fusion process for XRF consists of fusing a lithium borate and a powder sample, e.g., Cements, Limestones, etc. into a glass bead. For this, platinum ware (crucible and mould) is used.

The crucible filled with the powder sample and lithium borate is placed on a ceramic support inside the VITRIOX® ELECTRIC for then be taken into the furnace. During the deposition of a crucible inside the machine, errors such as the crucible handle being positioned in the wrong direction or a skew crucible with an axis not properly placed in its mounting may occur. To avoid that a crucible in the wrong position enters the furnace, these errors should be detected automatically through image analysis methods.

During the period of 01.04.2021 to 30.09.2021, research was made to compare which image analysis method was more suitable to detect the crucible position [2]. Three methods of computer vision were utilized. One was based in depth image processing algorithms though a custom software called 'QCTyche' while the other two were focus on machine learning algorithms, one using the deep learning model from the beh.cam [3] and one using a custom deep learning model configured specifically for the crucible position task. It was observed that customizing a deep learning model for the crucible position showed better results than the other two methods. The Beh.cam model showed difficulty in classify the correct position of the crucible and while the QCTyche software was able to correctly detect the position of the crucible in the majority of the tests, it showed a great dependence on the environment inputs (Light, distance of the camera to the crucible and color of background objects) [2].

Deep learning is a branch of machine learning that uses a deep network with numerous processing layers and multiple linear and non-linear transformations to model high-level abstractions of data. As the number of layers in a neural network increase, the trainable parameters will rapidly raise as well. This can make model training computationally intensive, especially for high dimensional data, e.g., images and videos. Making the tuning of so many parameters a massive undertaking.

Convolutional Neural Networks are fully connected feed forward neural networks that are widely used for training algorithms with image input data. These types of neural networks are exceptionally good at lowering the number of parameters without sacrificing the model quality, this is done by a convolution and pooling procedure that will be explained in the next chapter. Therefore, the Convolutional Neural Network is the main choice when it comes to image classification tasks.

To avoid that a wrongly placed crucible inside the furnace causes an accident, an image classification task will be used to determinate if the position of the crucible is feasible before entering the furnace. As stated above, the Convolutional Neural Network is the best choice for this task and will be the network type used for this thesis.

1.2 GOAL OF THIS THESIS

In this thesis, three different Convolutional Neural Network architectures will be trained and evaluated according with the crucible axis and handle position classification, each one tested for different image lighting. The main purpose of this thesis is to give an insight of the best model for the crucible position classification task.

The goal is to compare not only the accuracy but also the robustness of the models when the light in the picture changes form the one in the dataset used for training. A list of objectives can be seen bellow:

- Development and labelling of a custom crucible dataset that can be used in different Neural Networks.
- Configuration and training of different Convolutional Neural Networks models.
- Selection of the necessary camera and lens for implementation of the models.
- Development of a custom software to implement the different models.
- Evaluation and comparison of the trained models in different image lightning.

This thesis does not aim to discuss the history or steps involving in the creation of Convolutional Neural Networks, but in applying existing CNNs models and comparing their results for a specific task. The work of this thesis is limited to the VITRIOX® ELECTRIC machine crucible positioning, as well as the limitation of available computational hardware and limited time for creation of a custom dataset.

1.3 THESIS STRUCTURE

In the next chapter, the basics of machine learning are introduced first, so that the concept of Convolutional Neural Network can be presented right after. Chapter 3 The Dataset presents the development and preparation of the crucible dataset. After that, it follows the chapter containing the used methodology, with a discussion about how the selected CNN architectures were chosen and a brief explanation of their structure. Chapter 5 Experimental Setup explain the selection of the equipment used and the development of the software used for testing of the models. In Chapter 6 Evaluation and Results, the accuracy and error rate of all trained models is presented

and discussed. Finally, Chapter 7 Conclusion and Future Work shows the final thoughts about the obtained results, and brief recommendations for future research.

2 BASICS OF MACHINE LEARNING AND CONVOLUTIONAL NEURAL NETWORKS

The focus of this thesis is on machine learning techniques, particularly Convolutional Neural Networks. Despite the fact that it will not be able to go into great length on the subject, it's vital to have a basic understanding of it. First, the fundamentals of machine learning will be introduced and then move on to describe how convolutional neural networks works.

2.1 BASICS OF MACHINE LEARNING

The core concepts of machine learning will be discussed in this section, as well as the steps involved in training a basic artificial neural network.

2.1.1 WHAT IS MACHINE LEARNING?

Machine learning is an area of artificial intelligence (AI) that focuses on using data and algorithms to mimic the way that a human brain learns. It applies machine learning algorithms to the provided data in order to "learn" how to perform a specific task, and when talking about data, it's in reference to a variety of value types: images, numbers, texts, etc., essentially, any digitally saved data can be utilized as a dataset for machine learning.

According with Goodfellow et al. [4] the learning process can be divided in the three following species.

- **Unsupervised learning:** The machine learning dataset contains unlabeled data, which means that the input contains examples with several features but has no label; the process consists of learning the main properties of the dataset structure. Common unsupervised learning algorithms are the ones that use clustering, which consists of dividing the datapoints into clusters with similar properties.

- **Supervised learning:** Each datapoint in the dataset has a corresponding label that identifies its content. The learning process consists of generating a "rule" that translates the input to the corresponding output label. In the crucible dataset used in this thesis, each image was labelled as "Good" or "Bad" according with the crucible position displayed in the picture.

- **Reinforcement learning:** The learning algorithm has a feedback loop that feeds its interactions with the environment back to the learning system, each interaction receives a reward or punishment according with the desired outcome. Reinforcement learning are commonly used in games and robotics.

On this thesis the focus will be on supervised learning, and each datapoint on the crucible dataset will be labelled accordingly. There are two main strategies in supervised learning: *classification* and *regression*.

Classification: In this technique the system predicts in which of the given classes or labels the incoming datapoint belongs to. Object detection is an example of a classification task in which the input is an image, and the output is a number that represents the object's class.

Regression: In the regression technique, the systems receive a collection of independent values x as input and a collection of output values y that are depending on each input, the system then generates an equation f that best fits the relation $y = f(x)$. A simple example of a regression is a Fahrenheit to Celsius conversion where it's fed to the system a collection of Fahrenheit values as input and their corresponding Celsius temperatures as output, the function created by the learning algorithm should be general enough to predict the Celsius temperature for any Fahrenheit value.

As stated in the title of this thesis, the task is to use machine learning to classify a crucible position, for this reason, going forward the focus will be only on classification technique.

2.1.2 ARTIFICIAL NEURAL NETWORKS

Artificial neural network (ANN) is a subset of machine learning, its research has a long history and has evolved into a large and interdisciplinary field of study.

An ANN is a network of smaller computing units known as neurons. The ANN is often organized in layers. Each layer has a distinct number of neuronal units. The units are a simplified version of biological neurons, and every unit is linked to other units. The network is called a recurrent neural network if it has connections to the same layer or back. The network is called a feedforward neural network if the link to the output layer is solely in one direction, since Convolutional Neural Network (CNN) is a specific type of Feedforward neural network, it is the one that will be presented on this thesis. In Figure 1, it's an example of a feedforward neural network.

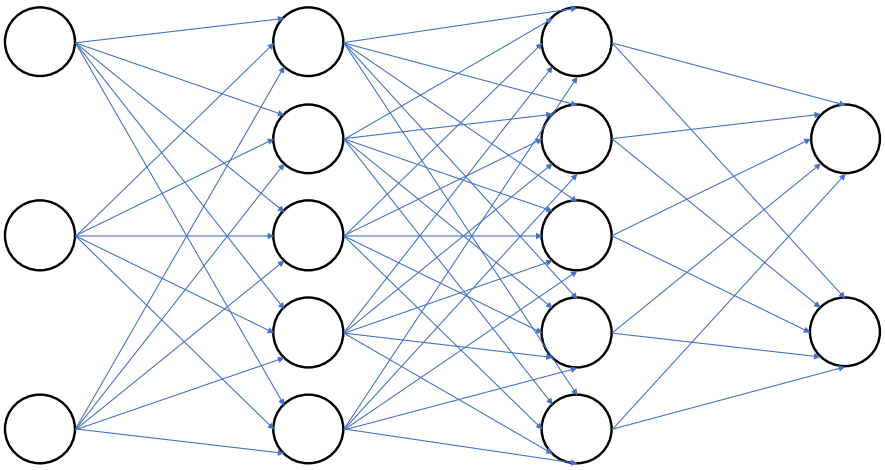


Figure 1: Feedforward neural network diagram.

For better understanding, bellow it's presented the definition of a *Perceptron*, first introduced by Rosenblatt [5].

Perceptron: The perceptron can be described as the simplest form of a feedforward neural network. It consists of two layers of neurons or units; a simple perceptron diagram can be seen in Figure 2.

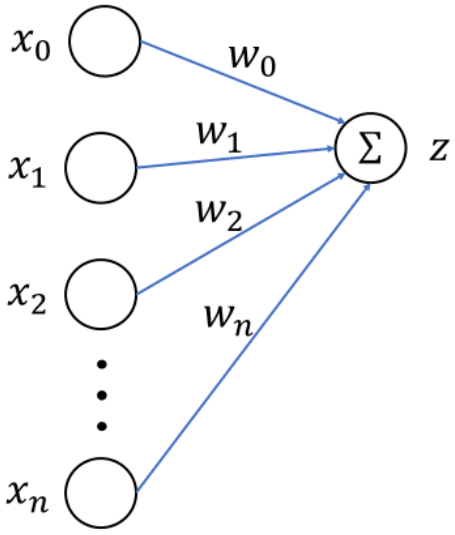


Figure 2: Base diagram of a perceptron.

Let $\boldsymbol{X} \in \mathbb{R}^{\boldsymbol{n}}$ be the input vector, where n is the number of features and $\boldsymbol{W} \in \mathbb{R}^{n}$ the weights. The output z of a perceptron can be calculated by a linear combination of each input $(x_0, x_1, x_2, \dots, x_n)$ with the weights $(w_0, w_1, w_2, \dots, w_n)$.

$$z = \sum_{i=0}^{n} x_i \cdot w_i$$

Eq. 1

During training, the weights are the parameters that will be learned to approximate the output z to the desired value. To a more complete perceptron model, it is inserted an activation function $\sigma(z): \mathbb{R} \to \mathbb{R}$, this is used to introduce non-linearity to the learning model. Furthermore, a bias term b is also introduced (Figure 3), the bias is an additional parameter that adjust the activation function to the left or to the right.

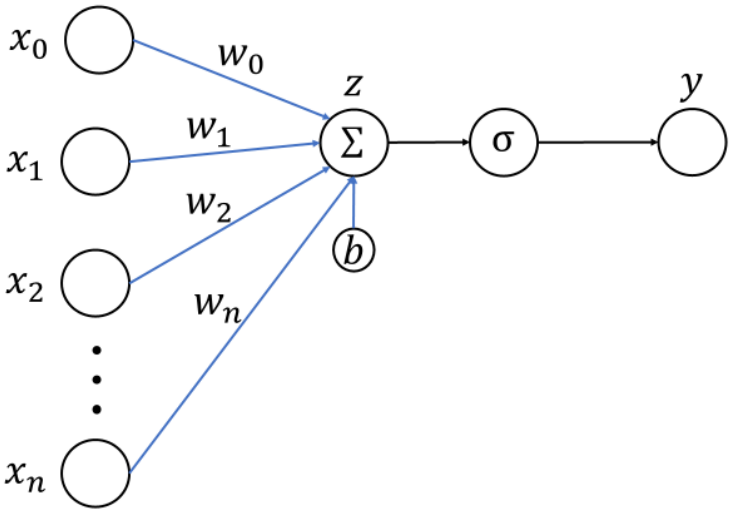


Figure 3: Perceptron diagram.

The output can be calculated as follows:

$$y = \sigma(z) = \sigma(\boldsymbol{X}\boldsymbol{W}^T + b)$$

Eq. 2

A multilayer feedforward neural network architecture will now be studied. For simplicity the first layer where the data features are received will be called as input layer I, the last layer where the predictions occur will be the output layer Z, all the layers in between will be denoted as hidden layer H. A multilayer feedforward neural network diagram is shown in Figure 4.

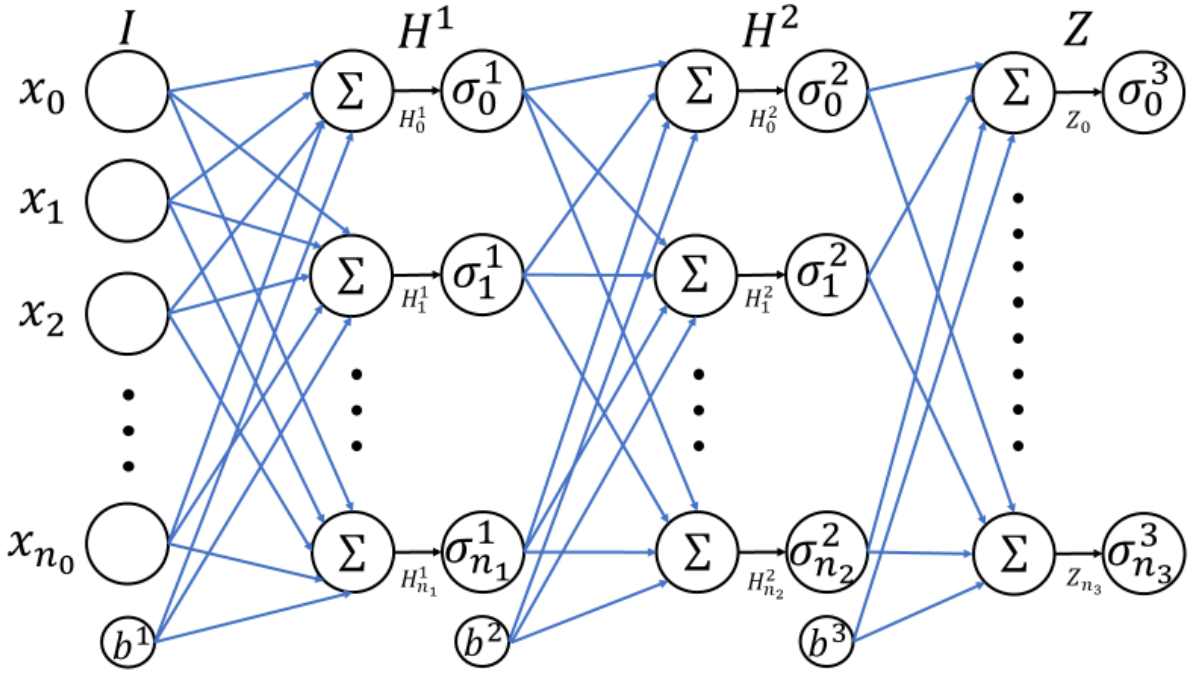


Figure 4: Multilayer feedforward neural network diagram.

Each layer is construct by one or more neurons, the number of neurons in each layer can differ, in other words, $n_0 \neq n_1 \neq n_2 \neq n_3$. Each connection (blue line) represents a weight or bias that will be updated during training using *backpropagation* that will be discussed in the next section.

2.1.3 THE TRAINING PROCESS

In this section the supervised training process of a multilayer feedforward neural network will be described, the topics discussed in this section are taken mainly from the following four books: Habibi Aghdam and Jahani Heravi [6], Mou and Jin [7], Zhou [8] and Skansi [9] where these topics are discussed in detail. As showed before, in the supervised training the output/label is known for each input, the objective of the learning algorithm is to find a function $f(\boldsymbol{X}, \theta)$ that better maps the input x_n to its corresponding label. Where $\boldsymbol{X}$ represents the input vector and θ are the learnable parameters. For that, a loss function will be used.

The loss function is used to determine whether the model got the result properly and, if not, how far way the model is from the correct output. In this thesis a common loss function called mean square error (MSE) will be used. It has the following formula:

$$J(\theta) = \frac{1}{2n}\sum_{i=0}^{n} |f(\boldsymbol{X}_i, \theta) - Y_i|^2$$

Eq. 3

Where Y_i is the desired output for the input vector $\boldsymbol{X}_i$ and θ is the weights that will be updated with the help of the loss function $J(\theta)$, for that, the algorithm uses the *gradient descent* method to minimize the loss function.

$$\theta_{new} \leftarrow \theta_{old} - \gamma \frac{\partial J}{\partial \theta_{old}}$$

Eq. 4

The learning rate γ determines the step size of each gradient adjustment, when training a machine learning model, gradient descent is an optimization approach. It's based on a convex function that iteratively adapts its parameters to get a given function to its local minimum. The gradient of the loss function in respect to the parameters θ needs to be calculated first so that the gradient descent can be updated.

The backpropagation is simply a way to compute the gradient $\frac{\partial J}{\partial \theta}$ more efficiently. For understanding the backpropagation better, the multilayer feedforward neural network from Figure 4 where it has an input and output layer together with two hidden layers it's shown again in Figure 5, but for simplicity, this time the input layer will contain two neurons, the first hidden layer H^1 will have three neurons while the second hidden layer H^2 will consist of two neurons, and the output layer will only have one neuron. The computational diagram is shown in Figure 5, for better visualization of the weights, the bias term is putted bellow each sum.

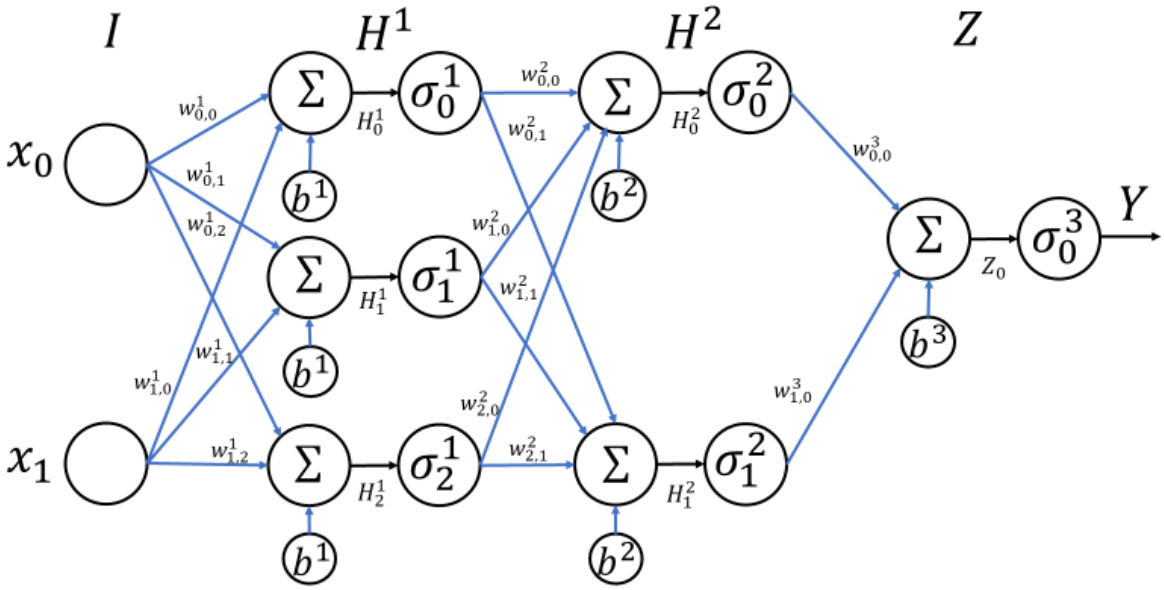


Figure 5: Computation diagram of the feedforward neural network example.

To calculate the gradient of the loss function J in respect to the weight $w_{0,0}^1$, all the other learning parameters are considered as constants, and the chain rule is used as follows:

$$\frac{\partial J}{\partial w_{0,0}^1} = \frac{\partial J}{\partial \sigma_0^3} \cdot \frac{\partial \sigma_0^3}{\partial Z_0} \cdot \frac{\partial Z_0}{\partial \sigma_0^2} \cdot \frac{\partial \sigma_0^2}{\partial H_0^2} \cdot \frac{\partial H_0^2}{\partial \sigma_0^1} \cdot \frac{\partial \sigma_0^1}{\partial H_0^1} \cdot \frac{\partial H_0^1}{\partial w_{0,0}^1} +$$

$$\frac{\partial J}{\partial \sigma_0^3} \cdot \frac{\partial \sigma_0^3}{\partial Z_0} \cdot \frac{\partial Z_0}{\partial \sigma_1^2} \cdot \frac{\partial \sigma_1^2}{\partial H_1^2} \cdot \frac{\partial H_1^2}{\partial \sigma_0^1} \cdot \frac{\partial \sigma_0^1}{\partial H_0^1} \cdot \frac{\partial H_0^1}{\partial w_{0,0}^1}$$

Eq. 5

That can be written in the factorial form:

$$\frac{\partial J}{\partial w_{0,0}^1} = \frac{\partial H_0^1}{\partial w_{0,0}^1} \cdot \frac{\partial \sigma_0^1}{\partial H_0^1} \left[\begin{array}{l} \left(\frac{\partial H_0^2}{\partial \sigma_0^1} \cdot \frac{\partial \sigma_0^2}{\partial H_0^2} \left(\frac{\partial Z_0}{\partial \sigma_0^2} \left(\frac{\partial \sigma_0^3}{\partial Z_0} \cdot \frac{\partial J}{\partial \sigma_0^3} \right) \right) \right) + \\ \left(\frac{\partial H_1^2}{\partial \sigma_0^1} \cdot \frac{\partial \sigma_1^2}{\partial H_1^2} \left(\frac{\partial Z_0}{\partial \sigma_1^2} \left(\frac{\partial \sigma_0^3}{\partial Z_0} \cdot \frac{\partial J}{\partial \sigma_0^3} \right) \right) \right) \end{array} \right]$$

Using Eq. 1 and Eq. 2:

$$H_j^k = \sum_{i=0}^{n_0} x_i \cdot w_{i,j}^k + b$$

Eq. 6

$$\sigma_j^k = \sigma(H_j^k)$$

Eq. 7

Where k is the current hidden layer level, j is number of the neuron where the signal is sent, while i is the number of the neuron where the input signal comes from. Is important to notice that this only calculates the gradient in respect to one single parameter, the process must be repeated for every parameter in the network.

With the gradient $\frac{\partial J}{\partial \theta}$ calculated, the gradient descent will set new parameters to the system and this process will repeat until the machine learning algorithm finds the better solution with the minimum loss.

To have a general view, in Figure 6 is a simple activity diagram of the backpropagation process explained above. The system receives a dataset input $D = \{(\boldsymbol{X}_i, \boldsymbol{Y}_i) | i \in \mathbb{N}\}$, the learning rate γ and the activation function σ is also given. The condition mentioned in the diagram is linked to the condition used to stop the model before overfitting, that will be discussed in section 2.1.5 Under- and Overfitting.

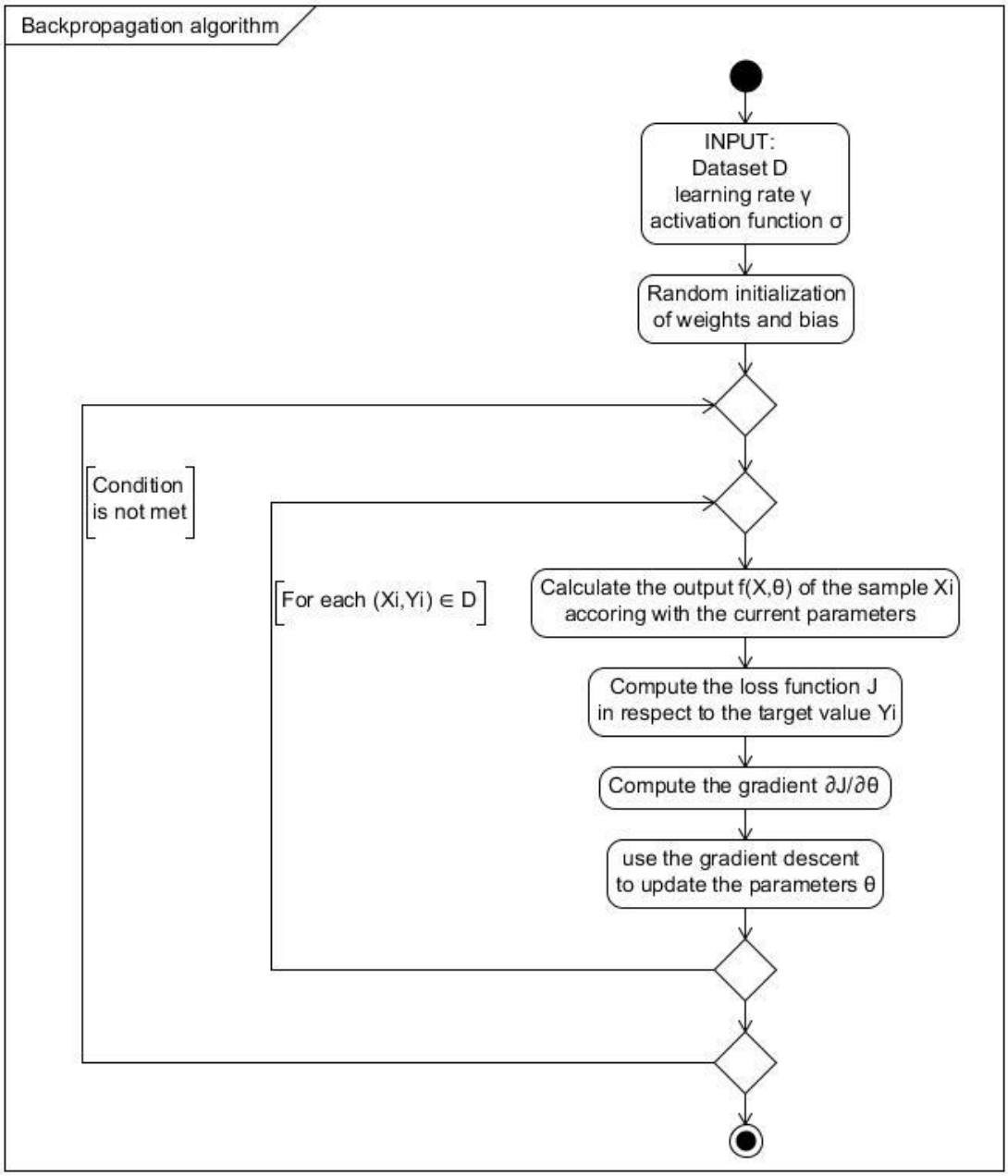


Figure 6: Backpropagation algorithm activity diagram.

2.1.4 ACTIVATION FUNCTIONS

The activation function σ mentioned on the previous section is an important part of a ANN. In neural networks, there are a variety of activation functions that can be used. Is good to understand that a feedforward network with linear activations in all neurons is nothing more than a linear function. So, at least one neuron with a nonlinear activation function is required to make a neural network nonlinear. For this reason, in this thesis it will mostly focused on nonlinear activation functions. Another important property of an activation function is that must

be continuously differentiable, so that the backpropagation algorithm is able to compute the gradient $\frac{\partial J}{\partial \theta}$.

Three main activations used in an ANN training are shown below. The sigmoid activation function has a working rage of $[0,1]$ while in the hyperbolic tangent activation function, the working range is $[-1,1]$. Both functions behave the same way for small values of z, the gradient value goes to zero, the learning convergence declines, and the neural network training takes a long time. Another issue appears when training deep neural networks (a neural network that contains many hidden layers), because backpropagation is frequently multiplied with small values, the vanishing gradient occurs in the early layers. For this reason, sigmoid and hyperbolic activation functions are suitable for shallow neural networks (with only a few hidden layers). The rectified linear unit activation function has a working range of $[0, \infty)$, a significant gradient is always produced in this region. As a result, it is free of the vanishing gradient problem.

Sigmoid: The sigmoid activations function is described as:

$$\sigma(z) = \frac{1}{1 + e^{-x}}$$

Eq. 8

The sigmoid function has the property to map any real value to the range $[0,1]$. The last layer of a machine learning model can use a sigmoid function to turn the network's output into a probability score, which is easier to work with and analyse. It has its name from the S-shape form from the function, as seen in Figure 7:

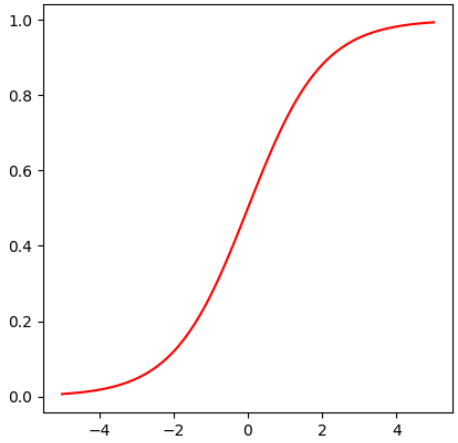


Figure 7: Sigmoid.

Hyperbolic Tangent: It works similar to the sigmoid activation function. Its formula is showed in Eq. 9.

$$\sigma(z) = \frac{e^{x} - e^{-x}}{e^{x} + e^{-x}}$$

Eq. 9

Like in the sigmoid function, the output values change rapidly when x approximates 0, but its working range is $[-1,1]$ and unlike the sigmoid function, close to the origin, the hyperbolic tangent function approximates the identity function. The values $\sigma(0) \approx 0$ and $\sigma'(0) \approx 1$ easy demonstrate this. The hyperbolic tangent function is shown in Figure 8

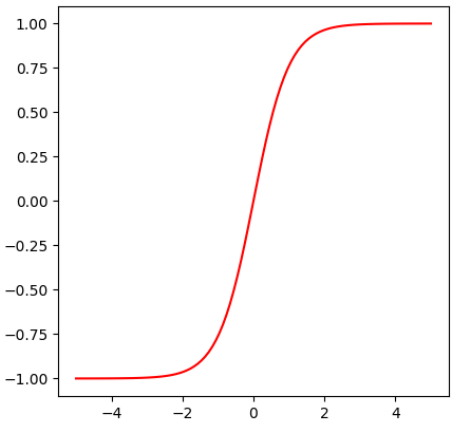


Figure 8: Hyperbolic Tangent.

Rectified Linear Unit (ReLU): It's the most used activation function for deep neural networks. It is defined as follows and is very efficient.

$$\sigma(z) = \begin{cases} 0 & z < 0 \\ z & z \geq 0 \end{cases}$$

Eq. 10

In simple words, the ReLU function eliminates the negative values. One of the characteristics of ReLU activation is that it might result in the death of neurons during training, this could happen when a neuron's weight has been altered so that the z value of the neuron is always negative. Furthermore, the function that must be inserted to get to ReLU is the max function $max(0, x)$, shown in Figure 9, which requires less computational power than sigmoid and hyperbolic tangent.

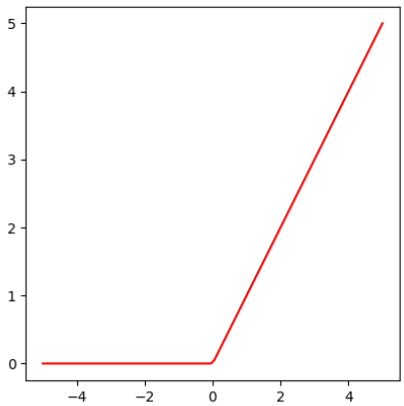


Figure 9: Rectified Linear Unit.

2.1.5 UNDER- AND OVERFITTING

A machine learning algorithm must be able to perform well with data that has never been seen before. Overfitting and underfitting are two ways that can cause the algorithm to perform poorly on unseen data. The learning model should generalize well the features from the training sample so that these features can be used to recognize any future new data. But when the model starts to learn too well, meaning, the model starts learning noise and specific features as general features, the model goes on overfitting and the general performance of the model decrease. Underfitting is the opposite effect, when the model only learns very broad features and is not able to recognize specific similarities on the samples. On the diagrams in Figure 10, the main principle of under- and overfitting are shown, where the black dots are the data samples and the red line represent the curve learned by the machine learning model.

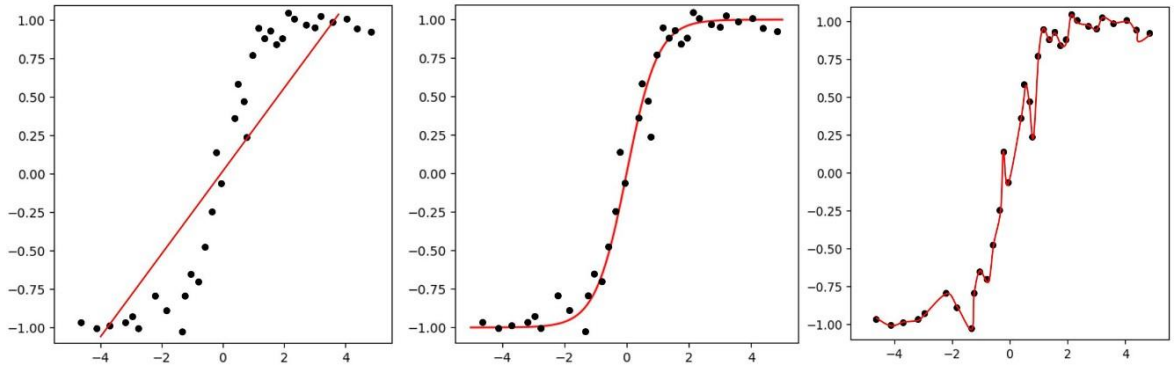


Figure 10: Under- and Overfitting diagrams.

The first diagram on the left shows an underfitting of the data resulting in a model that cannot recognized the right curvature of the function representing all data points. On the center is presented a generalized model that shows a curve that better represents the data points. On the right diagram, it shown a model that suffer of overfitting and thus it fits all the trained data points, but when used to handle new data points, it will be observed a bad *generalization error* (The ability to function well when confronted with unknown data).

Think on the example where it's fed to the model a dataset with a variety of horse images. An overfitting of the model would be if the model learned a specific feature of the horse, for example the color of the fur, as a general feature and then could not recognized any new sample were the horse color differ from the one that has been learned. On, the other side if the model is too general, as example, the model classifies any picture of a four-legged animal as a horse, that is a underfitting of the data.

While simpler functions are more likely to generalize (to have a small gap between training and test error), a suitably complicated hypothesis to obtain a low training error still need to be chosen [4].

In an ideal world, you'd pick a model that falls between underfitting and overfitting, but this is quite difficult to reach in the real world. Underfitting is rarely addressed since it is simple to discover. It can be solved by a better sampling of the dataset or simply by experimenting with

different machine learning techniques. Nonetheless, it serves as a good counterpoint to the issue of overfitting.

2.1.6 REGULARIZATION

As stated in the previous section, the ability of the model to perform well on unseen samples it's called *generalization error*. Regularization is simply any method used to reduce this generalization error. Goodfellow [4] talks about how there is no optimum machine learning algorithm or regularization method, the importance is in selecting a regularization technique that best match the problem that the model is attempting to solve. Three well known regularization methods are presented below.

L_2 Regularization: This method of regularization calculates the L_2 norm of the weights W and adds to the loos function J. This regularized loss function is defined as:

$$J_{L_2}(\theta) = J(\theta) + \lambda \|W\|^2$$

Eq. 11

The term $\|W\|^2$ is the L_2 norm, it serves to penalize the weights in the loss function. The term λ is a hyperparameter that defines how severe this penalization will be. The value of the loss function increases when added the regularization term, as a result the values of the weights matrices decreases. Assuming that a neural network with fewer weights produces simpler models, the overfitting is reduced to a certain extent. On the other side, if λ is too high, the model becomes too simple and enters on underfitting. For this reason, the choice of the λ value must be done very carefully.

Batch Normalization (BN): Develop by Ioffea and Szegedy [10], they introduce the term *internal covariate* that refers to the fact that the distribution of each layer's inputs changes during training since the values of previous layers parameters also changes. This makes it infamously difficult to train models with saturating nonlinearities because it necessitates lower learning rates and careful parameter initialization. Batch normalization is intended to solve this problem by applying normalization for each training step, or as is called in [10], *mini-batch*. The BN transform is showed below.

$$\mu_B \leftarrow \frac{1}{n}\sum_{i=0}^{n} x_i$$

Eq. 12

$$\delta_B^2 \leftarrow \frac{1}{n}\sum_{i=0}^{n} (x_i - \mu_B)^2$$

Eq. 13

$$\widehat{x_i} \leftarrow \frac{x_i - \mu_B}{\sqrt{\delta_B^2 + \epsilon}}$$

Eq. 14

Where $\mu_B$is the mini-batch mean, δ_B^2 is the mini-batch variance and ϵ is a constant added for numerical stability. $\widehat{x_i}$ represents the normalized input value, the batch normalization is applied after the linear combination thus:

$$z_i \leftarrow w\widehat{x_i} + b \equiv BN(x_i)$$

Eq. 15

Where w and b are the learned parameters of the model. Notice that the bias term b is already normalized by $BN(x_i)$. Finally, Eq. 2 can be rewritten as:

$$y = \sigma\big(BN(\boldsymbol{X})\big)$$

Eq. 16

Dropout: It's a regularization method develop by Srivastava et al. [11] that aims to prevent overfitting. When performing backpropagation, it creates brittle co-adaptations that work for the training data but fail to generalize to new data. The main idea of the dropout method is that for each round of the training, units will be dropped at random. This random dropout disrupts these co-adaptations.

Dropout has the disadvantage of prolonged training time. Due to the fact that not all weights are trained in a single iteration, since obtaining a response for a single sample necessitates repeatedly running the network.

2.2 CONVOLUTIONAL NEURAL NETWORKS

Convolutional Neural Networks (CNNs) is a subtype of artificial neural network (ANN) and it work similar to a standard ANN. The main difference is that a CNN is developed to learn patterns from high-dimensional input data, e.g., videos, images, or audios. CNNs uses high-dimensional filters, also called *kernels*, to perform convolution with the layer's input. This is the *Convolutional layer*; it's done to extract the main features of the input data and generate what is call *feature maps.*

After the convolution, the model perform a down sampling technique called pooling, it takes the feature maps resulting from the convolution and reduces the spatial size of the given input. This is done on the so-called *Pooling layer*.

A convolutional neural network is composed by an input layer followed by a series of convolutional and pooling layers. Then, at the end the fully-connected layers are inserted. The fully-connected layers will then attempt to produce class scores from the activations, which will be utilized for classification, in the same way that normal ANNs do.

In Figure 11, a diagram of a standard CNN is shown to better understanding. In the next two sections, the convolution and pooling layers will be discussed in further details.

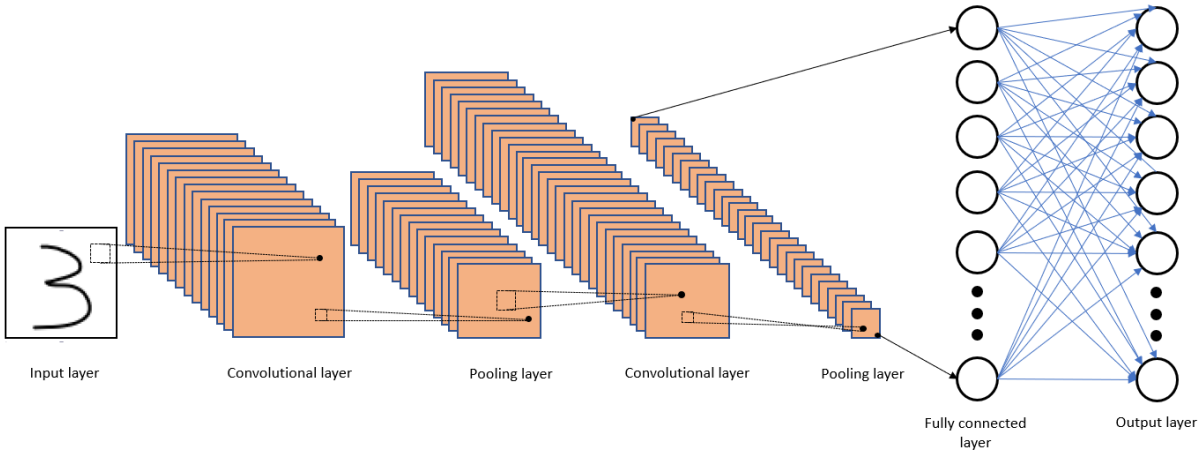


Figure 11: CNN basic architecture.

2.2.1 CONVOLUTIONAL LAYER

As described before, the task of the convolutional layer is to extract the main features of input data. This is done with the help of filters/kernels that will be optimized during training. A key difference between a CNN and an ANN is that while the input for a standard artificial neural network is a vector, for a convolutional neural network, the input is a tensor. In Figure 11 the input is a grayscale image, which means that the input is a tensor of size $W \times H \times 1$, where W and H are the width and height of the image in pixels. Now look at Figure 12 where it has a color crucible image, the image is digitally represented as a three-channel image, for example, a RGB image type. The input will be a tensor of size $W \times H \times 3$, where the tensor depth represents the red, blue, and green channels of the image.

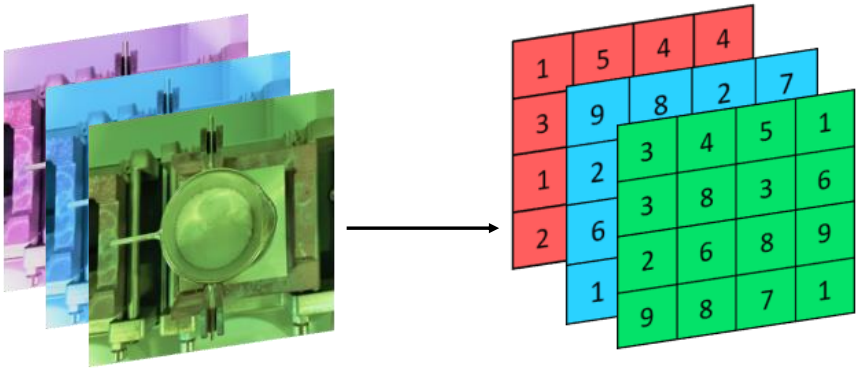


Figure 12: Digital representation of a RGB image.

The convolution procedure consists of a dot product of the kernel with each channel input. By passing one kernel though one input channel will generate a single feature map. If the input has three channels, for each kernel utilized, three feature maps will be generated. The next layer will have $k \times n_i$ channels, where k is the number of kernels that will be used and n_i is the number of channels of the previous layer.

Now, a kernel of size 2×2 is selected and passed though the green channel of the coloured input image example from Figure 12. A common way to set the initial values of a kernel is by random initialization, then of course, the kernel will be updated as the model is trained. A simple representation of a convolutional procedure is shown in Figure 13.

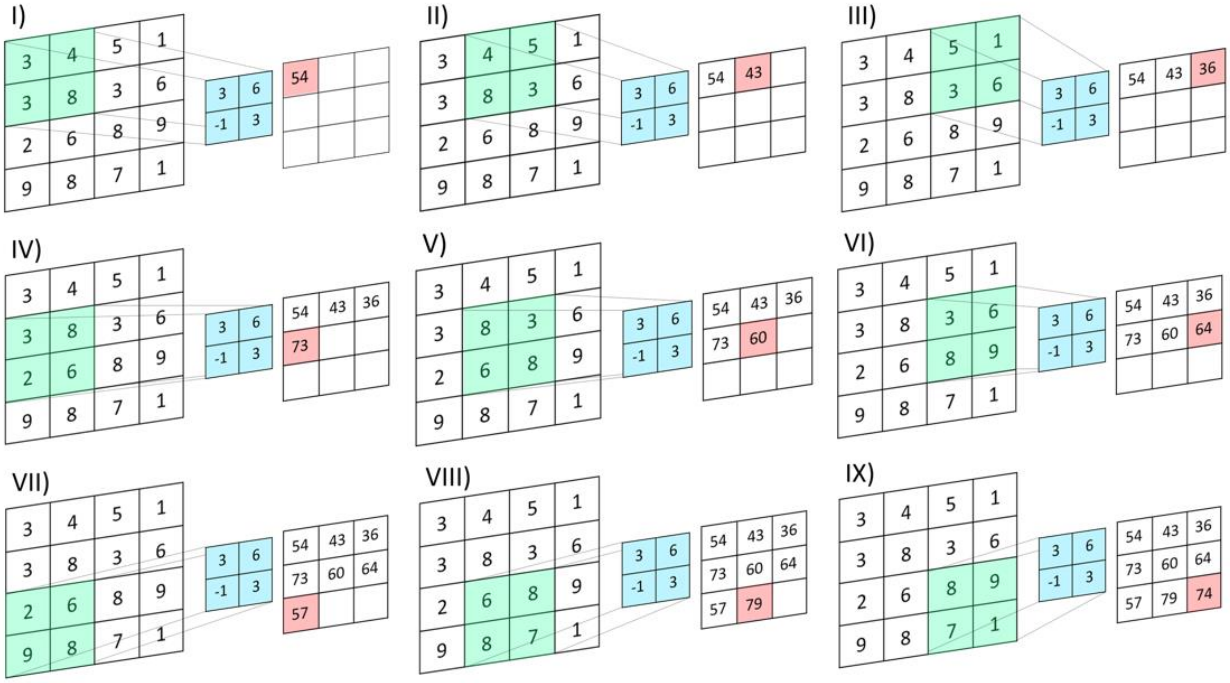


Figure 13: Convolution procedure example.

Observe that the feature map has a smaller dimension then the input data. A common method to avoid that is to use a technique called *zero-padding*, it's a basic process of padding the input's boundary with the value 0. This helps to prevent that any important information from the input data is left out and gives more control over the output dimensions.

Aside from the number of filters and kernel size, the stride is another parameter for a convolution layer. The stride is the number of steps taken during the sliding across the input channel. In the case showed above, the stride is $(1,1)$, and the local connectivity for the next output unit is to slide one unit to the right or down. If the stride is raised to $(2,2)$, the next output unit is calculated by sliding the local connectivity two units to the right or down.

2.2.2 POOLING LAYER

The pooling layer is added after convolution, the goal is to reduce the spatial size of the feature maps that were generated by the previous layer. By down sampling the data, the training time is reduced, and the model requires less computational power. The two main utilized pooling methods are *max pooling*, that get the maximum value of the current filter field. And *average pooling*, which calculates the average value. In Figure 14 is shown an example of max and average pooling. A filter of size 2×2 is being used in a 4×4 feature map, with a stride of $(2,2)$.

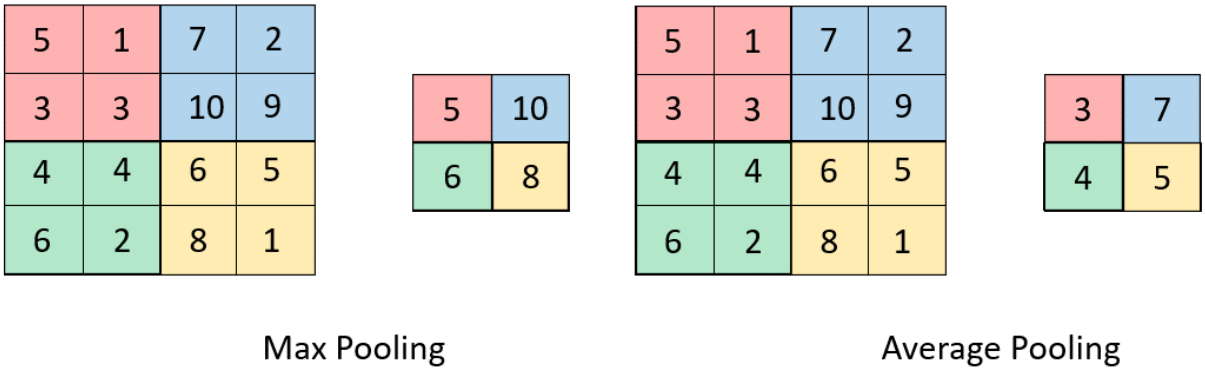


Figure 14: Max and average pooling.

It's important to mention that pooling is done independently for each feature map. If for example, the previous convolutional layer has 40 feature maps of size 8×8 and it's applied pooling with the size 2×2 and stride of $(2,2)$. The pooling will be done in each of the 40 feature maps and will result on a layer of dimension $4 \times 4 \times 40$.

2.3 TRANSFER LEARNING

Transfer learning it's a popular technique in machine learning. It's usually uncommon to have available a big enough dataset for your training, and to start the training with random initialized weights can increase a lot the training time. An alternative is to perform transfer learning from a model that was already trained with millions of images.

There are two typical way that transfer learning can be performed. One is to transfer all or only part of a pre-trained model, then add more layers on top. The weights of the transferred layers will now be adjusted as well by continuing the backpropagation. It is feasible to keep some of the transferred model's bottom levels frozen.

The other approach is to use a pre-trained model as a feature extractor. A series of experiments showed that the generic features extracted from a convolutional neural network are very powerful and they are applicable to a variety of tasks and datasets (Yosinski et al. [12]).

The second approach is used in this thesis. A pre-trained TensorFlow model trained with the COCO 2017 [13] dataset was used. By using the first trained checkpoint, only the wights from the first layers are transferred and since the COCO 2017 dataset is very large and diverse, the

generic learned features e.g., corners, basic geometric shapes, etc are used as an advantage for the model. Without starting the training with random weights, the training time is reduced since the convergence of the model takes less time.

3 THE DATASET

The dataset used as input is a crucial aspect of every machine learning model. As mentioned in the previous chapter, in this thesis, a pre-trained model that was trained on the COCO 2017 [13] dataset was used. This dataset contains more than 200.000 labelled images with 1.5 *million* of object instances. Examples of sample images for the COCO 2017 dataset can be seen in Figure 15.

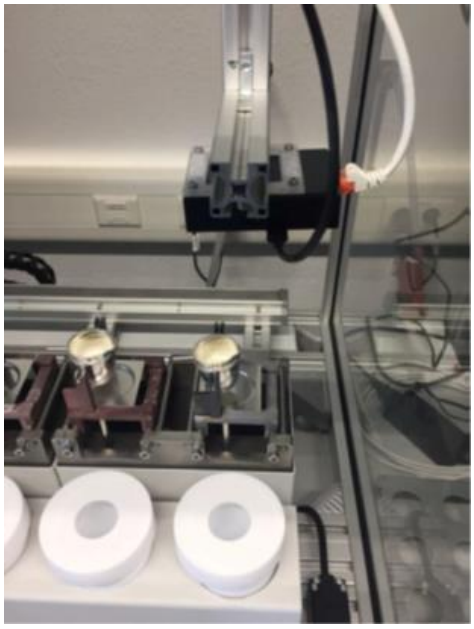
Figure 15: Samples of labelled images in COCO dataset.

To fine tune the pre-trained model to the task of crucible position classification, a dataset with crucible images needs to be created. The new model will use the initial weights of the pre-trained model and train the algorithm according to this new crucible dataset. The technique for gendering and preparing the crucible pictures for training will be described in the next two sections.

3.1 DATA ACQUISITION

The acquisition of the crucible images was mainly done with the Beh.cam [3] previously mounted in a VITRIOX® ELECTRIC [1] machine. Beh.cam is AI camera that uses its own software to machine learning training. For this thesis, the Beh.cam was only used for the acquisition of crucible images to be used with a pre-trained CNN model.

During the period of two weeks, every time that a crucible was prepered with a sample to be puted in the machine, several pictures of the crucible in different positions would be taken before the start of the fusion process. The mouting of the Beh.cam inside the VITRIOX® ELECTRIC machine for image acquisition is shown in Figure 16.



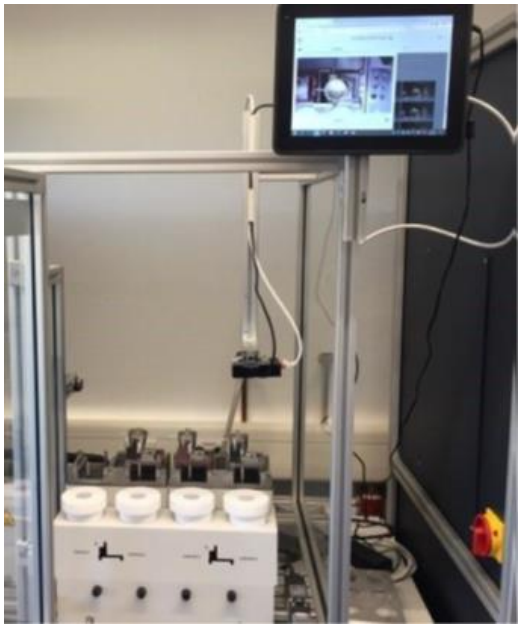


Figure 16: Mounting of the beh.cam inside the machine.

Many different samples with different colors are used inside the crucible for fusion, as well different ceramic supports for the crucible. For this reason, all pictures were then converted from RGB images to grayscale images. This way, the difference in color from the samples and the crucible support will not influence in the training of the model.

Besides the Beh.cam pictures, 23 images were taken by an iPhone, to be able to capture crucibles in a different slot in the machine. At total, 260 images of the crucible were taken.

All the obtained images were then separated in two classes: Good and Bad. The “Bad” images are the ones were the handle or axis of the crucible are placed in a wrong way in the crucible support while the “Good” images are the ones where the handle and axis of the crucible are placed correctly in the support. Figure 17 and Figure 18 displays the difference between a good handle position and a bad handle position.

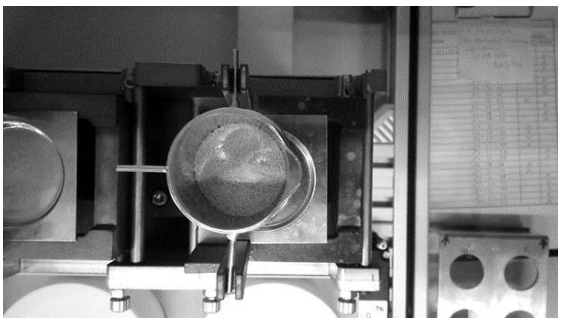


Figure 17: Good handle position.

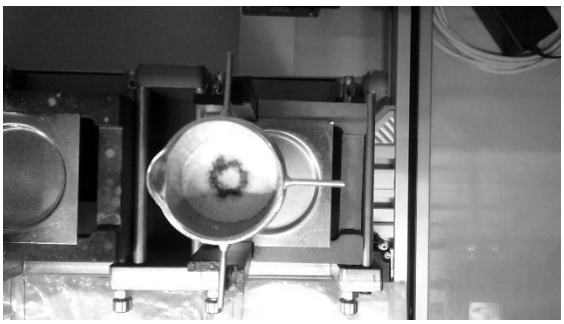


Figure 18: Bad handle position.

The handle should not be placed above the platinum mould plate, since after the crucible exits the furnace, the fusion mix is poured into the mould by the autosampler with an auxiliary rod as seen in Figure 19.

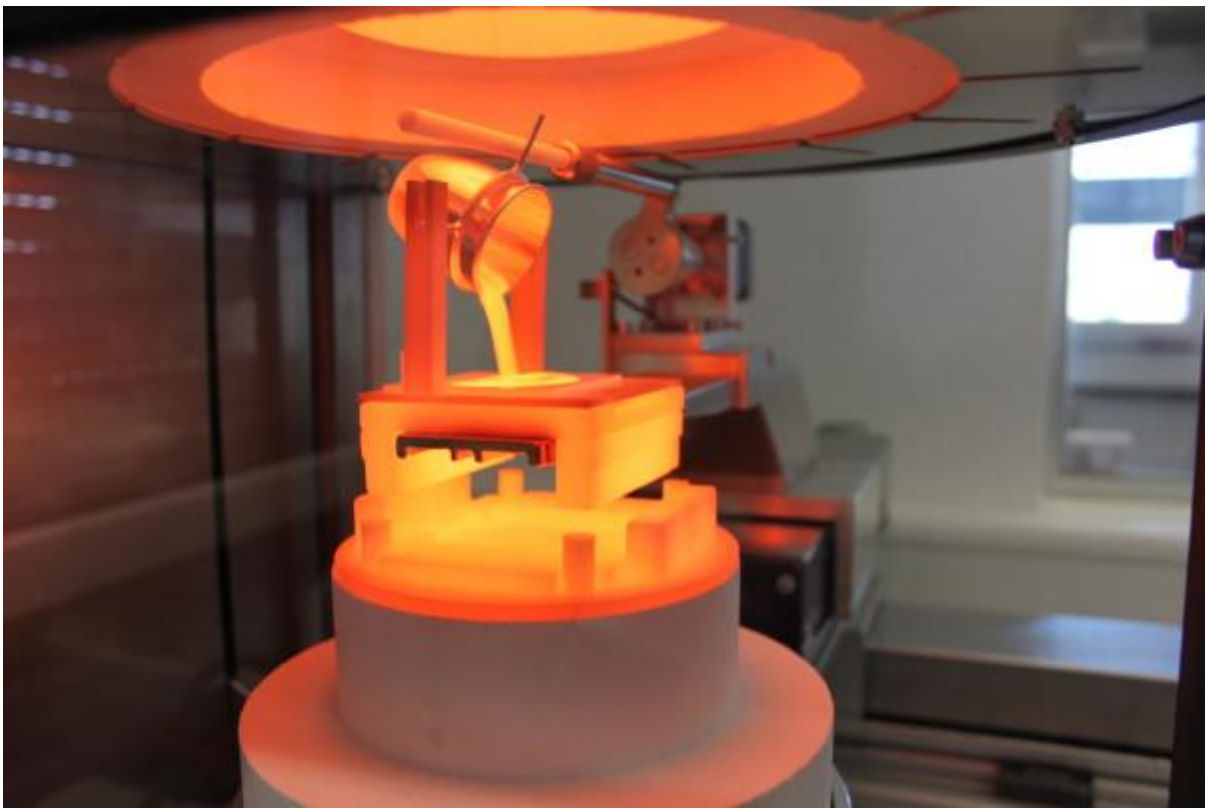


Figure 19: Pouring of the homogenous melt into the mould.

For the same reason, the axis must be placed in the support's slit as seen in Figure 20, if it placed outside the slit, the crucible would fall out of the support during the rotation of the sample inside the furnace causing an accident.

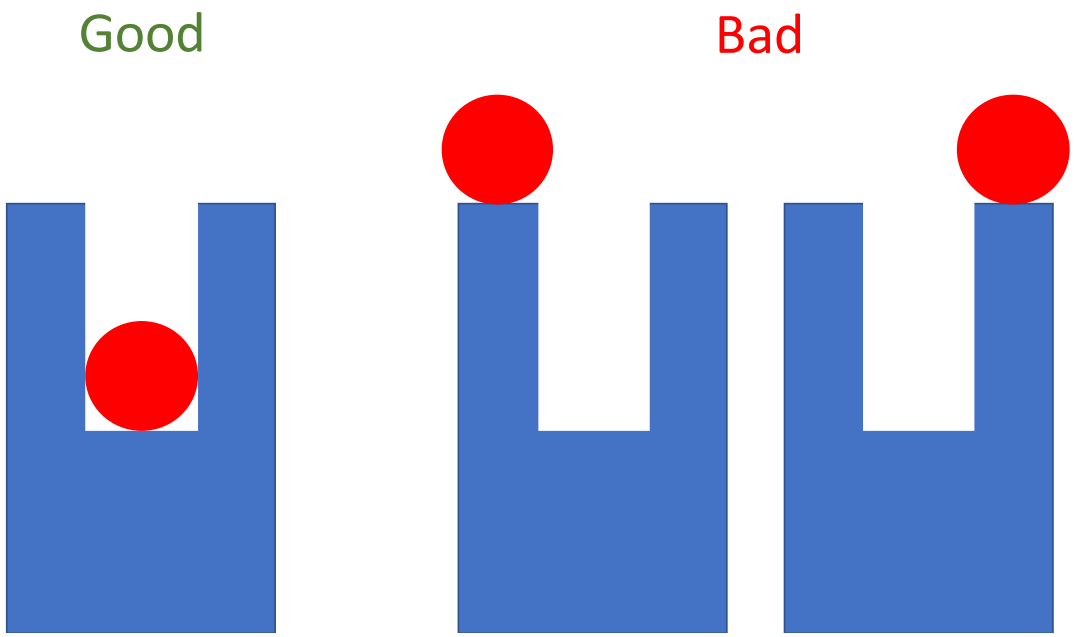


Figure 20: Difference between a good and bad axis position side view.

Figures 21-26 shows the difference between a good axis position of a bad one.

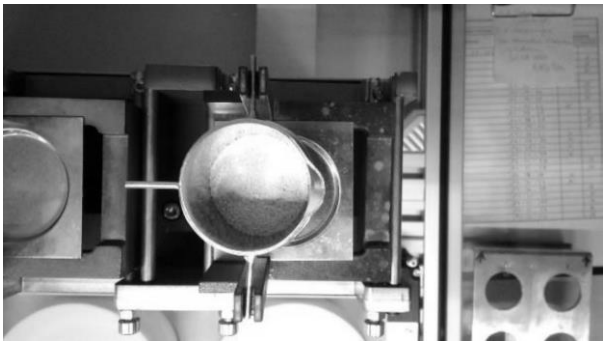


Figure 21: Good axis position example 1.

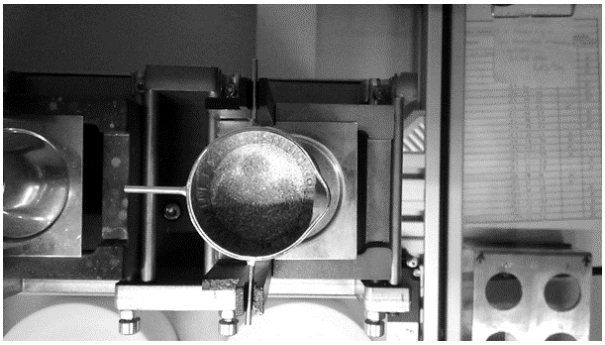


Figure 22: Good axis position example 2.

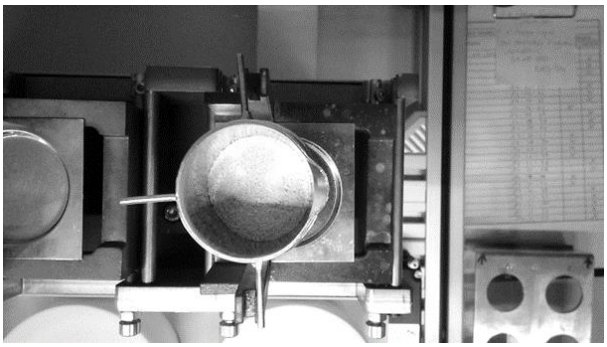


Figure 23: Bad axis position I.

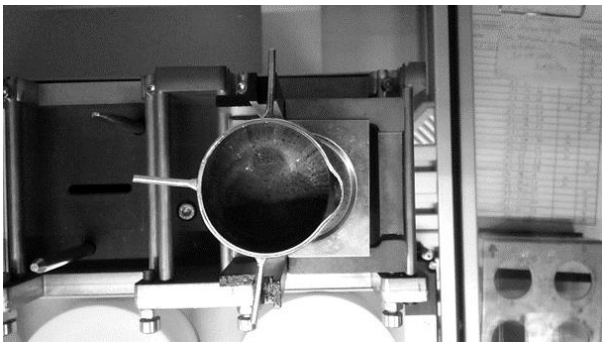


Figure 24: Bad axis position II.

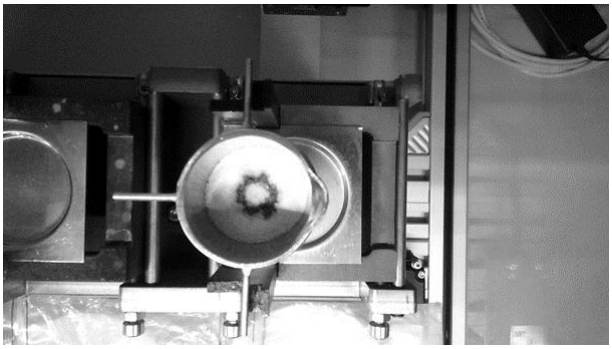


Figure 25: Bad axis position III.

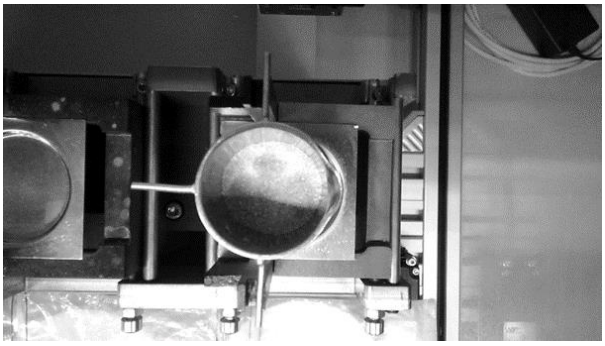


Figure 26: Bad axis position IV.

I. Bad axis position 1: crucible shift on the anticlockwise direction.

II. Bad axis position 2: crucible shift on the clockwise direction

III. Bad axis position 3: crucible shift to the left

IV. Bad axis position 4: crucible shift to the right

3.2 LABELLING OF THE IMAGES

For labelling the crucibles images, a software called LabelImag [14] was used. It's a free graphical image annotation tool. Each image was labelled one by one with "Good" or "Bad". The LabelImag software creates an XML file for every image indicating the labels in the picture and their location. 142 images were utilized for training the model for classifying the handle position of the crucible. Figure 27 and Figure 28 demonstrates the process from labelling the handle position.

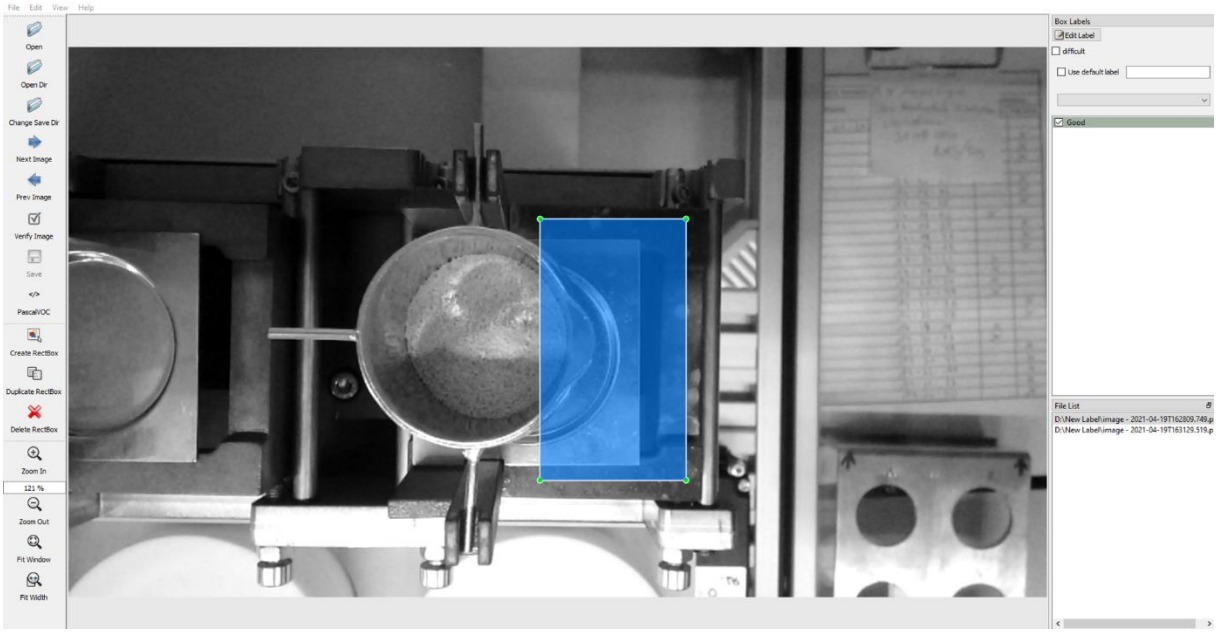


Figure 27: Labelling of a good handle position.

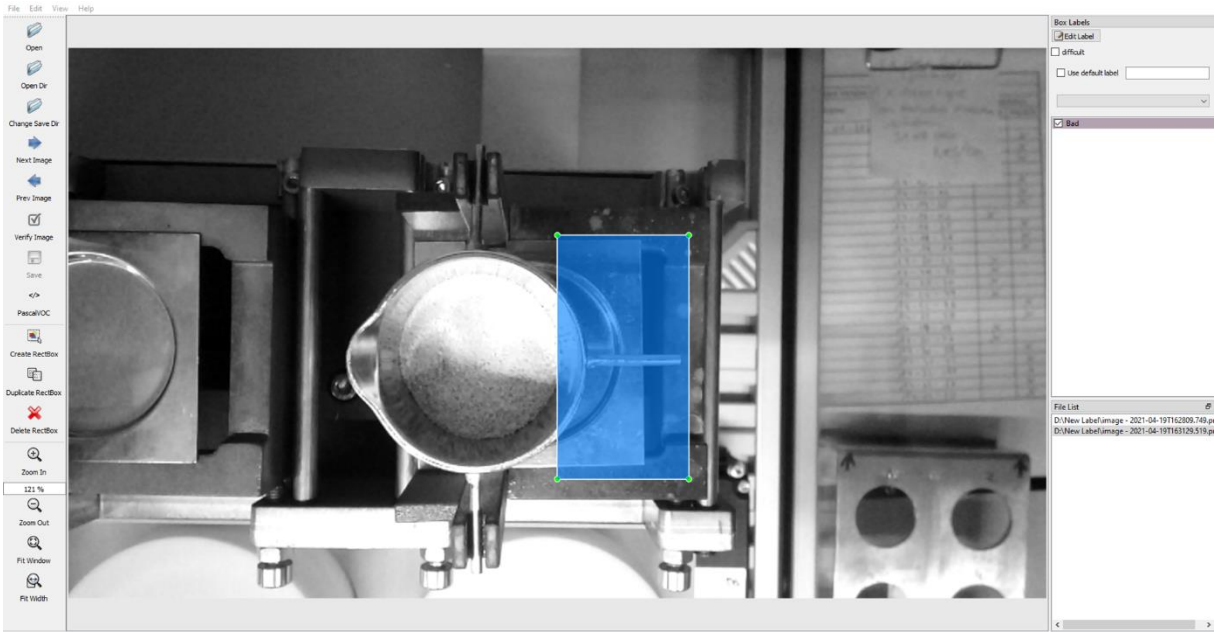


Figure 28: Labelling of a bad handle position.

For the axis position classification training, 180 images were used. Figure 29 and Figure 30 shows the labelling process of the good and bad axis position of the crucible. In each picture, the labelling occurs in both axes, the upper and lower one.

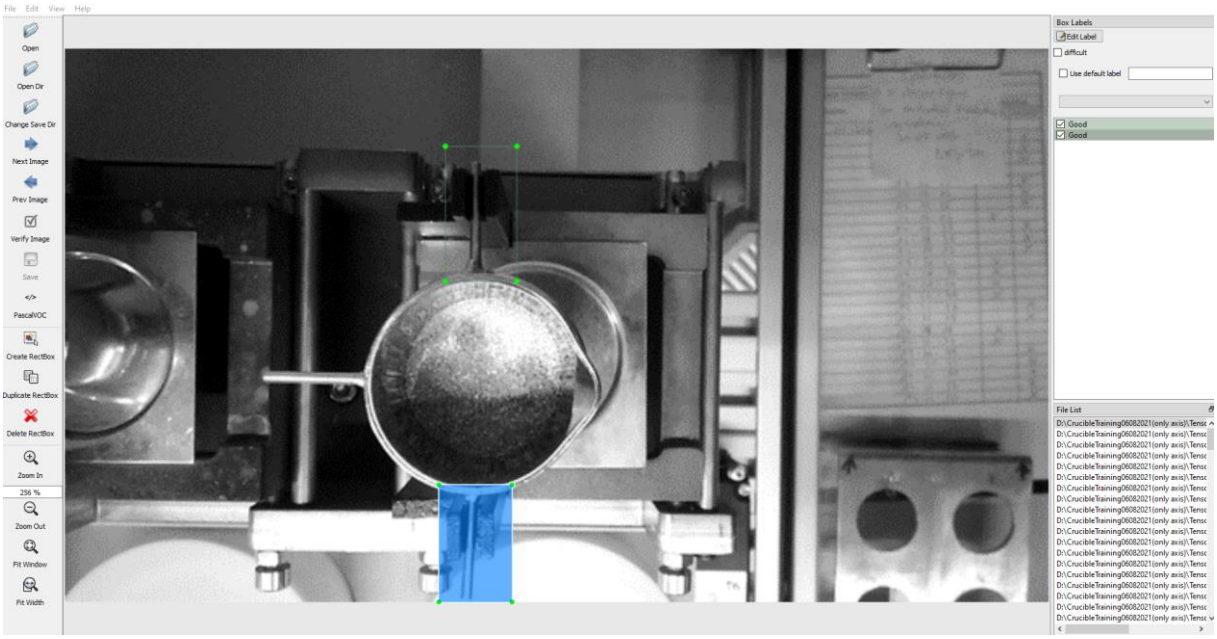


Figure 29: Labelling of a good axis position.

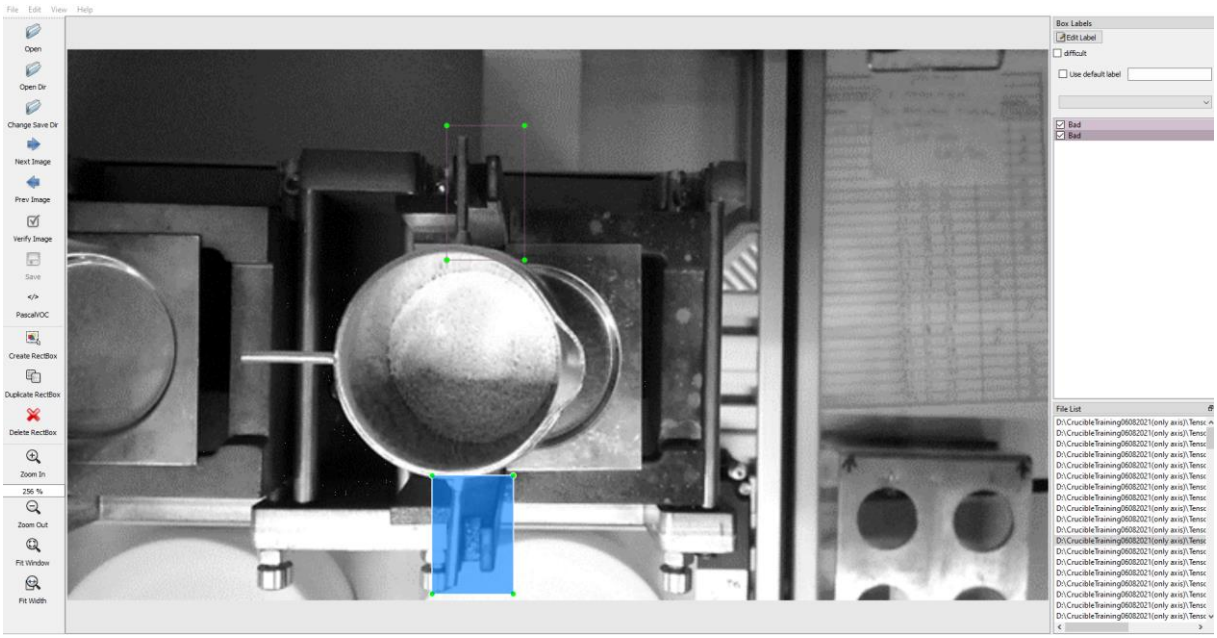


Figure 30: Labelling of a bad axis position.

The images and their corresponding XML files should be in the same folder location and have the same name, only differing in the extension file type. Figure 31 shows an example of a label XML file, notice that under the size section, between $< size >$ and $</size >$, are the width, height, and depth of the picture, these are the dimensions of the tensor of the image in the Input layer of the model. Later on, in chapter 4 Methodology, It will show that all images resolution

is reduced to $320x320$ for calculation proposes, making all the input tensors have the size $320x320x3$.

The object section of the file, between $< object >$ and $</object >$, it's where the information of the label is stored. For the axis position, one label is placed on the upper axis and other label is placed in the lower axis. The name is the class of the object, in the crucible classification task, it can only be "Good" or "Bad". The xmin, ymin. xmax, ymax values are the boundaries of the label box.

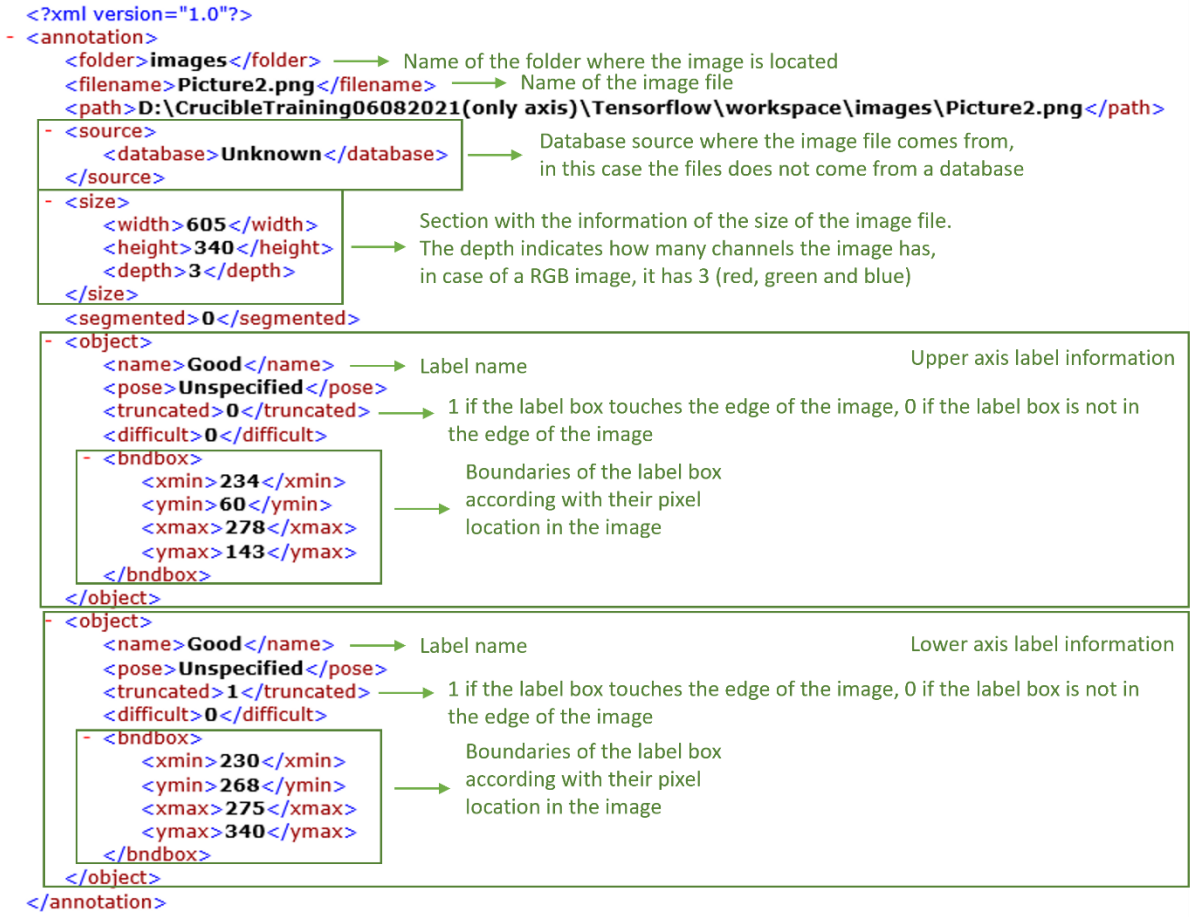


Figure 31: Label XML file example.

4 METHODOLOGY

As stated in chapter 2 Basics of Machine Learning and Convolutional Neural Networks, in this thesis, it will be used a pre-trained models download from TensorFlow 2 Detection Model Zoo [15]. In this chapter, the CNN architectures used for the crucible position classification task will be explained as well all the main libraries and software used for the training process.

4.1 IMPORTANT SOFTWARE COMPONENTS

Besides the LabelImag software used for the acquisition and labelling of the images mentioned in chapter 3 The Dataset. Other important software's used for the training of the models will be briefly discussed in this section to a better view of the training process.

4.1.1 TENSORFLOW

For training the CNN models, a machine learning library needs to be used. Machine learning has a big range of training libraries, e.g., Pytorch, TensorFlow, Theano, Caffe, etc. The library used in this thesis will be TensorFlow 2 [16]. It's an open-source library develop by Google Brain Team and it's one of the most used libraries for deep learning. TensorFlow 2 is written in C++, Python and CUDA and provides APIs (Application Programming Interface) for both Python and C++, in this thesis TensorFlow 2 is used with *Python 3* [17]. It accepts input data in the form of tensors, which are multi-dimensional arrays with greater dimensions. When dealing with enormous amounts of data, multi-dimensional arrays are very helpful.

TensorFlow was created using data flow graphs to solve large numerical computations. These data flow graphs consist of nodes, that represents the mathematical operations, and graph edges that represents the tensors that pass thought the nodes. TensorFlow supports computations on Graphical Processing Units (GPUs), as well as Central Processing Units (CPUs).

Keep in mind that using a CPU, the training typically needs a longer training time. Another TensorFlow advantage it's the possibility of using Google Colab. It is a free hosted Jupyter notebook that already has TensorFlow installed and can be used directly to train machine learning models using Google's GPU. TensorFlow also has a large collection of data and pre-trained models.

This thesis will also use TensorBoard [16], which is the visualization toolkit for TensorFlow, and it allows you to track and visualize several metrics parameters such as accuracy and loss of the model on training or validation.

4.1.2 KERAS

Keras it's an open-source library written in Python, and it's implemented directly in TensorFlow since 2017. It's a user-friendly, high-productivity interface for addressing machine learning tasks. It provides fundamental abstractions and building elements for designing and deploying high-iteration-rate machine learning systems [18]. TensorFlow uses Keras as a High-level API on the model's structure creation.

4.1.3 OTHER USEFUL PYTHON COMPONENTS

As mentioned in section 4.1.1 TensorFlow, The machine learning model used in this thesis will be trained using python API for TensorFlow. Python has a few packages that are commonly used for machine learning and in this section, it will be discussed the main two packages: *Jupyter Notebooks* and *NumPy*.

Jupyter Notebook [19] it's a web application that is open-source and enables you to create interactive python code. You can divide the code in different sections, also call cells, and run each cells separately. In this thesis, it will be used to write the python code for the training process in an organized form.

A well-known library for handling the processing of large multidimensional arrays it's the NumPy library. NumPy has a wide range of mathematical functions, as well as random number generators, linear algebra algorithms, Fourier transforms, etc [20]. It's a library used by TensorFlow and will be used for handling the input images as tensor during the testing phase.

4.2 THE BASE NETWORKS

An object detection model can consist of two parts, A base network and a detection module. The base network acts as a feature extractor, and the detection module takes these features as input to provide outputs that contain object classes and coordinates of the detected objects (including the height (h), width (w) and center (x, y) of the predicted box).

Huang et al. [21] wrote a guide for selecting an object detection architecture by comparing the accuracy and speed in different combinations of base networks and detection modules. Three detections modules were used: Single Shot Multibox Detection (SSD) [22], Faster Region-based Convolutional Neural Network (Faster R-CNN) [23] and Region-based Fully Convolutional Network (R-FCN) [24]. The base networks used with each detection module were VGG-16 [25], MobileNet [26], Inception V2 [10], ResNet-101 [27], Inception V3 [28] and Inception Resnet V2 [29]. Table 1 extracted from [21] shows the overall mAP (mean Average Precision) numbers of the architectures trained in both Minival and Test-dev datasets.

Table 1: Test-dev performance of the "critical" point along optimality frontier [21].

Model summary	Minival mAP	Test-dev mAP
(Fastest) SSD w/MobileNet (Low Resolution)	19.3	18.8
(Fastest) SSD w/Inception V2 (Low Resolution)	22	21.6
(Sweet Spot) Faster R-CNN w/Resnet-101, 100 Proposals	32	31.9
(Sweet Spot) R-FCN w/Resnet-101, 300 Proposals	30.4	30.3
(Most Accurate) Faster R-CNN w/Inception Resnet V2, 300 Proposals	35.7	35.6

In this thesis, six models (three for the handle position and three for the axis position) will be trained with the crucible dataset and compared. These models were derived from the following three architectures form Table 1 (with slily differences that will be presented in detail in section 4.4). The first one will be the fastest model: SSD with MobileNet. The second one will be an architecture in the "Sweet Spot", meaning, with a balance between speed and accuracy: Faster R-CNN with Resnet-101,100 proposals. Finally, the third one will be the most accurate architecture: Faster R-CNN with Inception Resnet V2. The three models with have the same image input resolution of $320x320$. Next, the base networks and detection models utilized for the crucible task will be briefly explain for better understanding of the results.

4.2.1 MOBILENETV2

MobileNetV2 [30] was introduced with the objective to be a lighter deep neural network to improve performance on mobile models. It archives that by using traditional convolutional only in the first layer and in other layers a *Bottleneck Residual Block* is used. Figure 32 shows the structure of these Bottleneck Bocks.

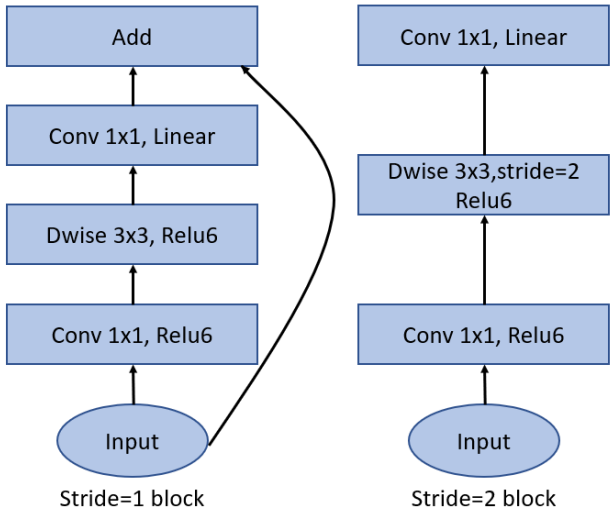


Figure 32: MobileNetV2 bottleneck blocks retrieved from [30].

As mentioned above, these convolutional blocks do not use normal convolutions but uses two techniques called *depthwise convolution* and *pointwise convolution*. The depthwise convolution uses a separated kernel in each input channel and performs each convolution separately, so unlike the standard convolution where the next layer has $k \times n_i$ channels, in a depthwise convolution, if the input image has three channels, the output will also have three channels. The pointwise convolution is similar to the standard one but uses a 1x1 kernel. The ReLU activation function is used for its good performance in calculations with low precision.

The overall architecture of the MobileNetV2 neural network can be seen in Figure 33. Notice that the stride s indicates which bottleneck block from Figure 32 will be used, the value c indicates the number of output channels of each block. This is an image representation of the information form Table 2: MobileNetV2 from [30]. This was made only for a better view of the architecture, for more information from each block, please refer to the table in the original document.

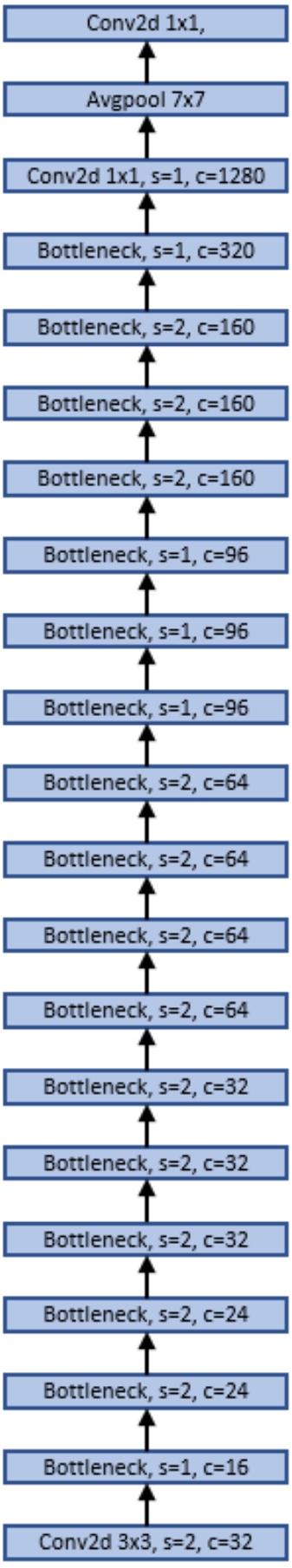


Figure 33: Overall view of a MobileNet architecture.

4.2.2 RESNET-101

ResNet-101 [27] is a residual Network (ResNet) and is one of the most powerful deep neural network architectures out there. The vanishing gradient problem mentioned in chapter 2 is of the main reasons for accuracy degradation and deep residual learning framework was created to overcome this problem. It lets each layer to intentionally match a residual mapping $F(x)$ rather than assuming that each few stacked levels will directly match a desired underlying mapping $H(x)$, where $F(x) \coloneqq H(x) - x$. Figure 34 shows a residual block of a deep residual network in comparison to a standard deep network.

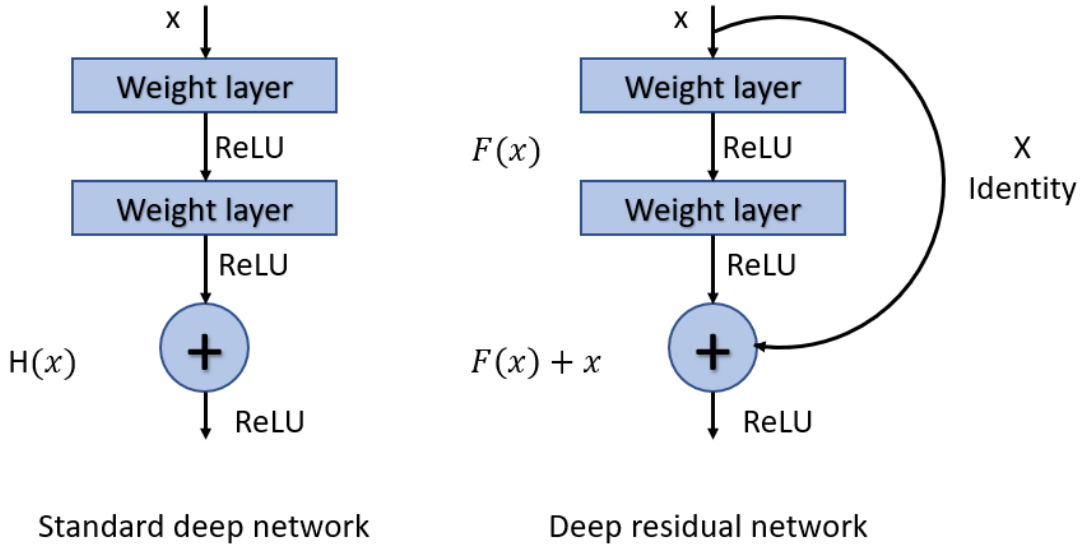


Figure 34: Comparison between a deep residual network and a standard deep network.

More simply, a deep residual network tries to learn $F(x) + x$, this helps to solve the vanishing gradient problem by allowing that the gradient signal travels back to earlier layer before it turns too small and insignificant, this happened thought these "shortcut connections". To a better overview of a ResNet-101 architecture, which has 101 layers, Figure 35 shows the building blocks that are repeated in the main architecture.

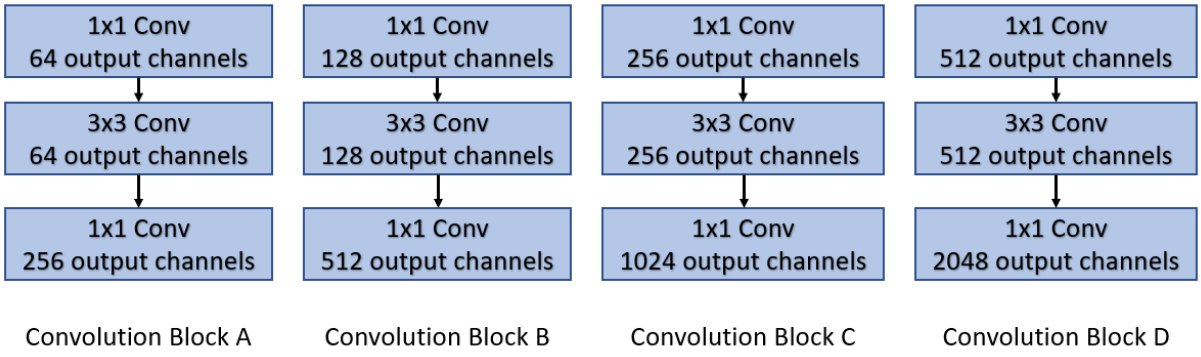


Figure 35: Convolution block for ResNet-101 [27].

The information for construction of these Convolution blocks are taken from Table 1 Architectures for ImageNet from [27] where a more detail description of each layer can be found.

The overall architecture for the ResNet-101 can be seen in Figure 36.

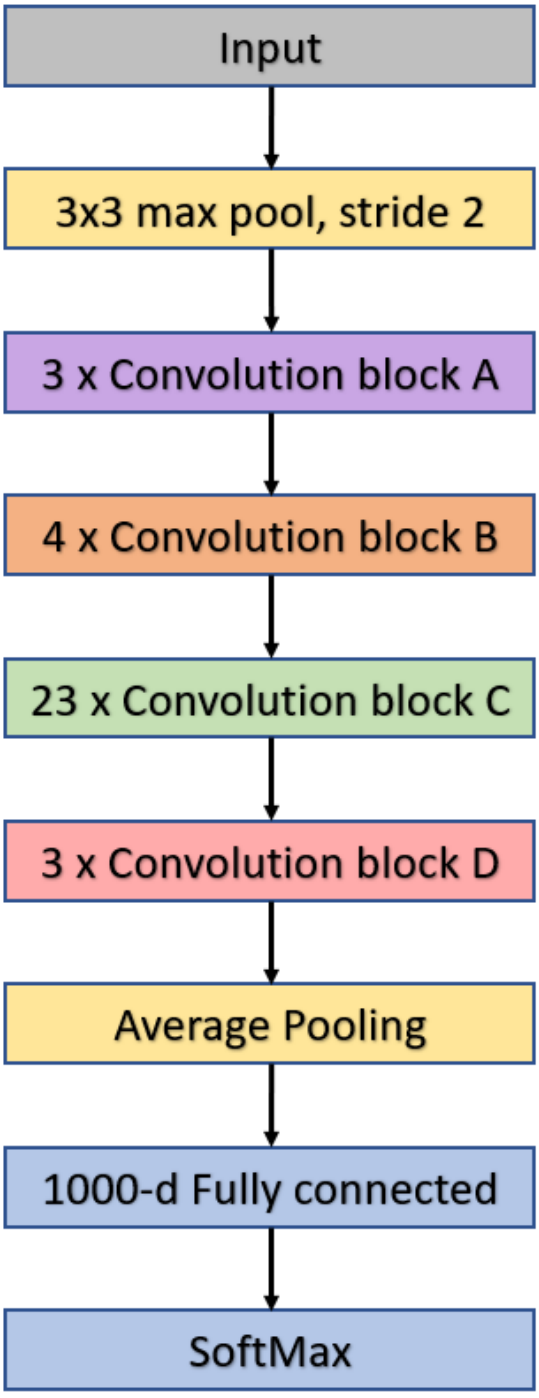


Figure 36: ResNet-101 architecture [27].

Again, this is only an image representation of the ResNet architecture information presented in Table 1 Architectures for ImageNet from [27] for a better general understanding. To see the complete information of ResNet and its variants, refer to the original document.

4.2.3 INCEPTION RESNET V2

Inception Resnet V2 [29] is a combination of the Inception and ResNet architectures. It's built in the base of an Inception architecture but also includes residual connections. This is done by adding a filter-expansion layer to the Inception blocks. Each Inception block is followed by a 1x1 convolution without activation. This is done to scale up the filter bank's dimensionality, so the reduction caused by the Inception block is compensated. The three Inception Residual block utilized on Inception ResNet V2 it's shown in Figures 37-39.

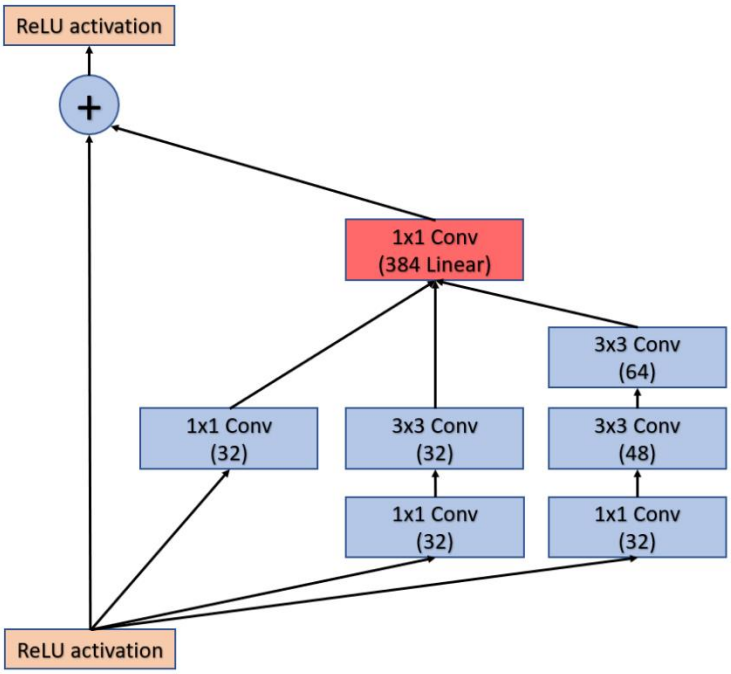


Figure 37: Inception-ResNet-A [29]; 35x35 grid Inception Residual block.

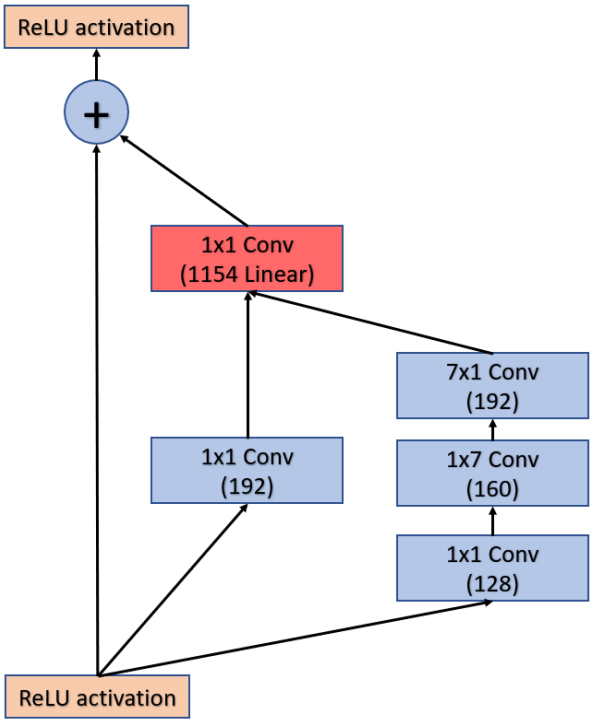


Figure 38: Inception-ResNet-B [29]; 17x17 grid Inception Residual block.

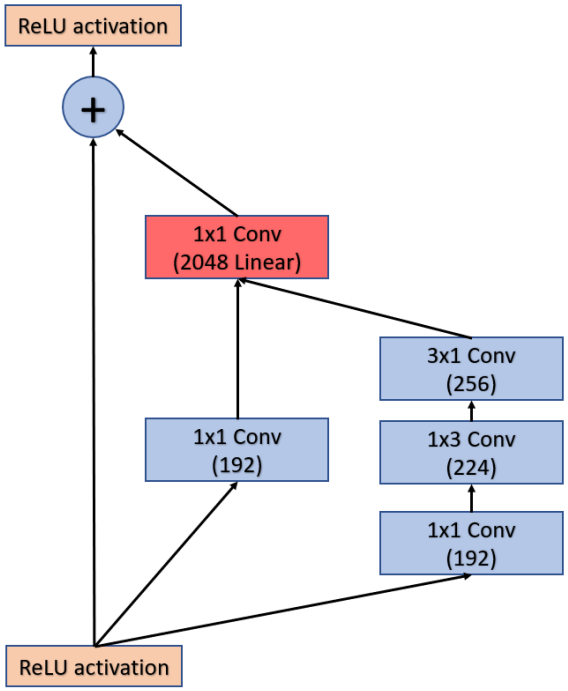


Figure 39: Inception-ResNet-C [29]; 8x8 grid Inception Residual block.

After every Inception-ResNet-A and Inception-ResNet-B, it's added a reduction block A and B respectively. According to Szegedy et al. [29], The training seems to be stabilized when the residuals were scaled down before being added to the prior layer activation. Scaling factors ranging from 0.1 to 0.3 were used to scale the residuals before adding them to the collected layer activations. The reduction modules used on Inception Resnet V2 are shown in Figure 40 and Figure 41.

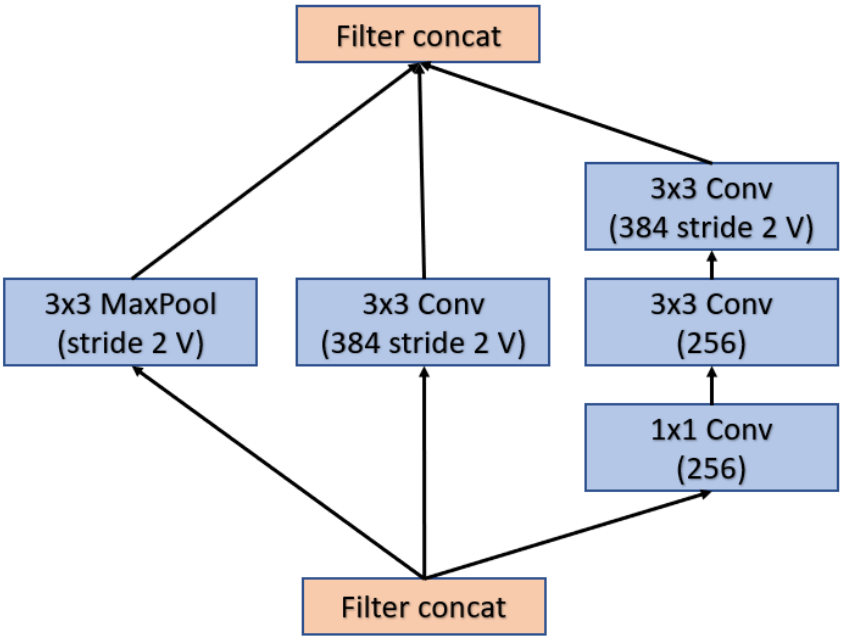


Figure 40: Reduction-A [29]; 35x35 to 17x17 reduction module.

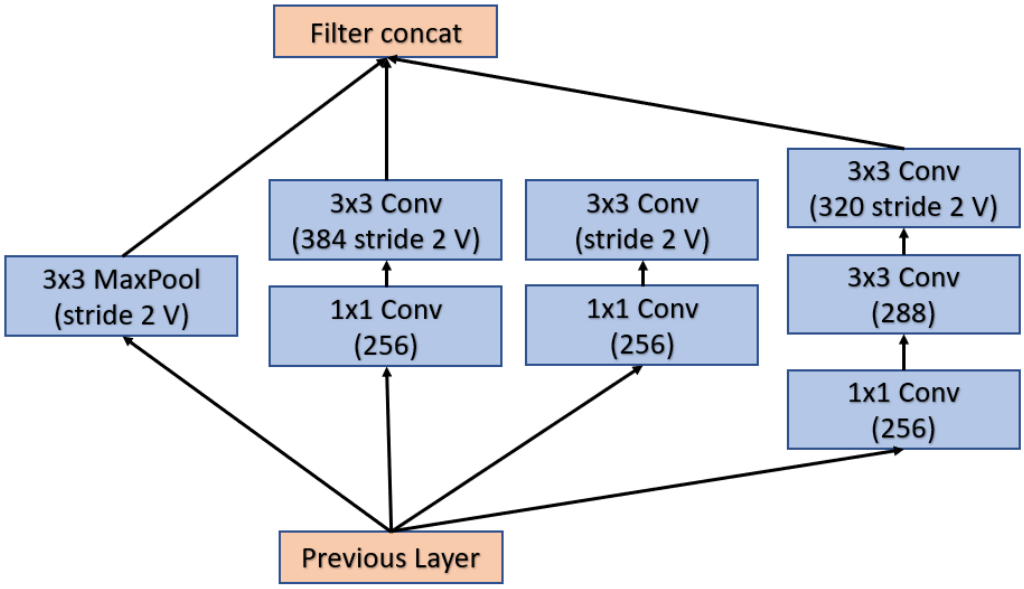


Figure 41: Reduction-B [29]; 17x17 to 8x8 reduction module.

The overall architecture for Inception ResNet V2 can be found in Figure 42.

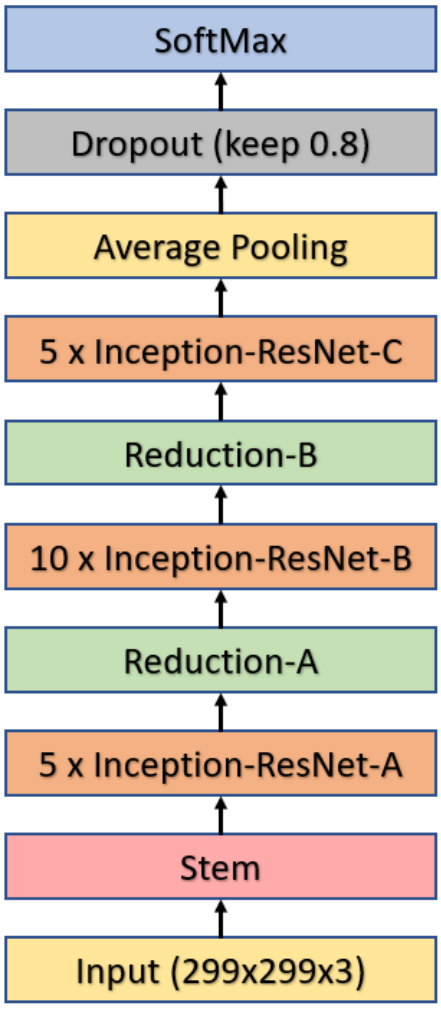


Figure 42: Schema for Inception ResNet V2 [29].

4.3 DETECTION MODULES

From the three pre-trained models used for this thesis, two of them uses the detection module Faster R-CNN and the other one uses SSD. In the next two sections, these detections modules will be described, to a better understanding of the complete models.

4.3.1 FASTER R-CNN

First, to start understanding Faster R-CNN [23] you have to look how a simple R-CNN (Region-based Convolutional Neural Network) [31] works. The principle behind a R-CNN it's the so-called *region proposals*. It extracts around 2000 bottom-up region proposals for every input image, then by using a large CNN, computes the features for each proposed region and classifies them by using class-specific linear SVMs. Performing a large Convolutional Neural Network for each region proposal make the training really slow and requires a lot of memory, for this reason, in 2015 Microsoft introduced the Fast R-CNN [32].

Instead of passing each region proposal trough a CNN, Fast R-CNN passes the entire image trough a set of convolution and pooling layers to generate a feature map. Then, for each region proposed a Region of Interest (RoI) is projected in this resulting feature map and then pooled through a pooling layer to a fixed-size feature map. The fixed-size feature map will then be mapped to a feature vector by passing through a set of fully connected layers (FCs). SoftMax probabilities and per-class bounding-box regression offsets are the network's two output vectors per RoI. A multi-task loss is used to train the architecture from beginning to end [23].

Faster R-CNN introduces Region Proposal Networks (RPNs) [23]. RPN takes an image as input and generates a set of rectangular region proposals. Faster R-CNN combines a RPN with a Fast R-CNN into a single network to make a detection module that is faster and achieve better accuracy scores. Figure 43 shows the architecture of a Faster R-CNN.

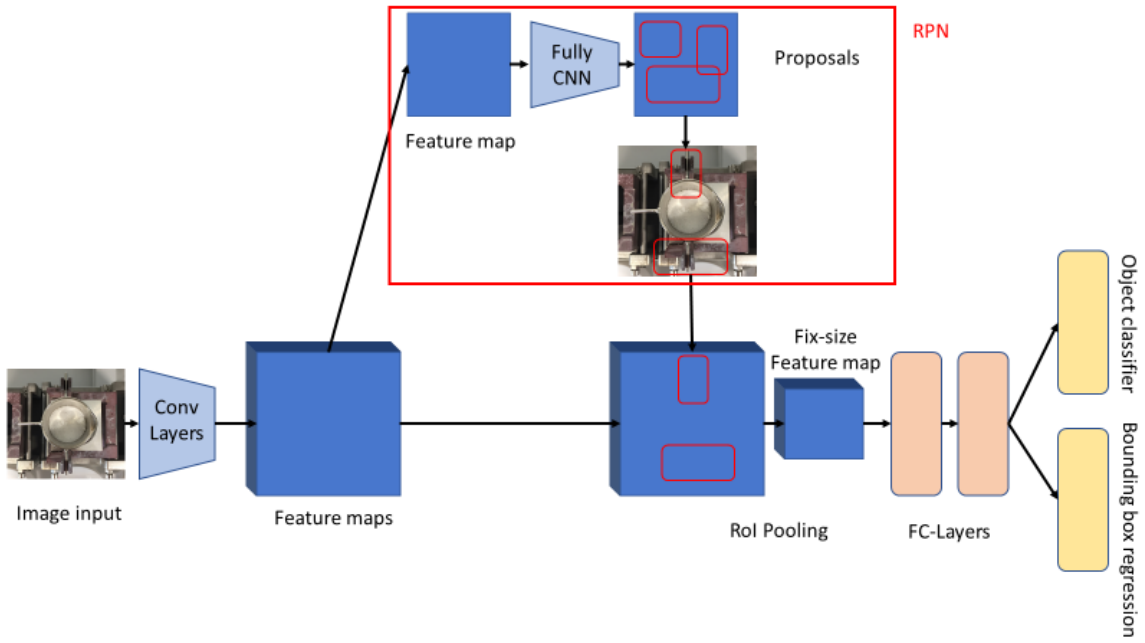


Figure 43: Faster R-CNN detection module.

Figure 43 was created to have a general overview of a Faster R-CNN detection module. It was constructed by garnering information of Figure 2: Faster R-CNN is a single, unified network for object detection from [23] and Figure 1. Fast R-CNN architecture from [32]. For better details refer to the original documents.

4.3.2 SSD: SINGLE SHOT MULTIBOX DETECTOR

SSD [22] was developed for real-time object detection. Opposite to Faster R-CNNs, SSDs does not utilise region proposals, which speeds up the process. SSD uses a technique called Multibox detection, that consists of evaluating a small collection of default boxes in multiple feature maps with varying scales. After the input image pass through a feature extractor, the resulting feature maps go through an added feed-forward convolutional network that generates a fixed-size collection of bounding boxes with occurrence scores of the object class in those boxes.

In simple words, SSD gets the key characteristics from the feature maps resulting from the base network (Inception, ResNet, etc.) and adds auxiliary structure with convolution filters to generate confidence scores for all object categories. Figure 44 shows a simple representation of an SSD detection module.

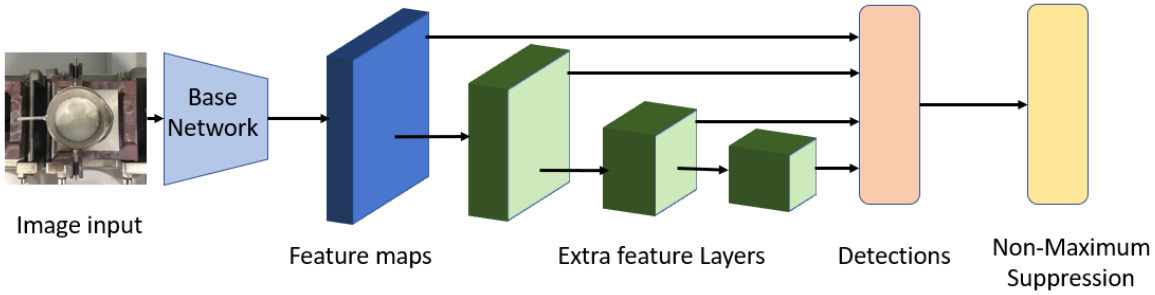


Figure 44: Simplified overview of an SSD detection module.

4.4 CONFIGURATION OF THE MODELS

As mentioned in section 4.2 The Base Networks, three model architectures chosen from Table 1 will be used. For each architecture, two models will be trained, one for the handle position and one from the axis position, resulting in a total of six trained models. Table 2 shows the description of each model.

Table 2: Model architectures description.

Handle position	SSD w/MobileNetV2; resolution: 320x320; batch size: 4
	Faster R-CNN w/ResNet-101, 100 Proposals; resolution: 320x320; batch size: 4
	Faster R-CNN w/Inception ResNet V2, 100 Proposals; resolution: 320x320; batch size: 4
Axis position	SSD w/MobileNetV2; resolution: 320x320; batch size: 4
	Faster R-CNN w/ResNet-101, 100 Proposals; resolution: 320x320; batch size: 4
	Faster R-CNN w/Inception ResNet V2, 100 Proposals; resolution: 320x320; batch size: 4

To train these models with TensorFlow, a set of libraries and software needs to be installed and configured. All the following steps and commands were made in a single Jupyter Notebook by following the Vladimirov guideline [33]. Here the commands used in the original training script are explained.

After the installation and setup of Python 3 and Jupyter Notebook on the computer, the next step is to install TensorFlow with GPU support. To more details on how to setup Python 3 on your machine, please refer to the guide [33] were detail information according with your operating system can be found, in this thesis the commands are for a Linux machine.

If you have Jupyter Notebooks working, all the following commands can be executed directly from the command cells without the need to open the command prompt, just create a new Jupyter Notebook in your project's folder. For installing TensorFlow, create a new command cell and execute the following pip command:

!pip install tensorflow-gpu

To be able to use the TensorFlow with GPU support, make sure that you have a GPU available on your machine together with the required CUDA libraries. The next step is to setup the TensorFlow Object Detection API [21]. First, download the 'models' folder from the TensorFlow repository on GitHub [34]. If you have Git [35] installed on your machine, simple use the following command directly in a new Jupyter notebook cell.

!git clone https://github.com/tensorflow/models.git

This will download all the files from the TensorFlow Model Garden to your local project folder. Before the TensorFlow Object Detection API can be used, a few dependencies need to be installed first. Protobuf is used by the TensorFlow Object Detection API to configure model and training parameters. First go to the Protocol Buffers release page [36] and download the latest release for your operating system. Then, extract the files downloaded, save in a location of your choice, and save this location on your environment paths.

Then, in you Jupiter Notebook, create a new cell and execute the following command to enter the research folder of the downloaded TensorFlow Model Garden.

cd {Your Project Path}/models/research

Where *{Your Project Path}* is the location of your project and where the files where downloaded. Then, to setup Protobuf, simply execute the following command in a new cell:

!protoc object_detection/protos/.proto --python_out=.*

The COCO API [13], it's also a dependency of the TensorFlow Object Detection API, for this reason, it also needs to be setup first. For that, the COCO API files need to be downloaded by executing the following command cell:

!git clone https://github.com/cocodataset/cocoapi.git

Then moving to the PythonAPI folder by the command:

cd cocoapi/PythonAPI

The output of this cell should be your current location: *{Your Project Path}/models/research/cocoapi/PythonAPI*. Then with the following command, you will build the COCO API:

!make

Finally, copy the 'pycocotools' folder to your research folder location with the following command:

cp -r pycocotools {Your Project Path}/models/research

After the Protobuf and COCO API are installed, the TensorFlow Object Detection API can be installed as well. For that, go back to the research folder location by executing the '*cd..*' command, if all the steps until here were followed correctly, this command should be executed twice, and the following output location should be.

{Your Project Path}/models/research

From this location, copy the TensorFlow Object Detection API setup file:

cp object_detection/packages/tf2/setup.py .

Then execute the following command to install the API.

!python -m pip install .

After that, the TensorFlow Object Detection API is ready to be used. Following right after, the configuration for the training of the models is to be made.

First, the pre-trained model needs to be downloaded from the TensorFlow 2 Detection Model Zoo page [15], the downloaded files contain the checkpoint that will be used for initialization of the new training and a configuration file *pipeline.config* where all the settings for the training of the model will be inserted. All these files are then copied to the same project location used in the previous steps, so the training script file is able to find all without problem.

After the download of checkpoints and config file for the desired model, the label map and the record files need to be created. The label map will tell TensorFlow what classes we want to detect, and the record files contain the information about all the label files created for each image. Figure 45 shows the created label map file.

```
item {
    name:'Good'
    id:1
}
item {
    name:'Bad'
    id:2
}

```

Figure 45: label_map.pbtxt

This label map can be created in the same Jupyter Notebook by creating a new cell and executing the following code:

```
labels = [{'name':'Good', 'id':1},
          {'name':'Bad', 'id':2}]

with open(ANNOTATION_PATH + '/label_map.pbtxt', 'w') as f:
    for label in labels:
        f.write('item { \n')
        f.write('\tname:\'{}\'\n'.format(label['name']))
        f.write('\tid:{}\n'.format(label['id']))
        f.write('}\n')
```

Where ANNOTATION_PATH is the location where you wish to save the file. For creating the record files, the python script 'generate_tfrecord.py' [37] was used. This python script generates record files from the XML documents created by the LabelImag software. For that, the following Jupyter Notebook cell needs to be executed:

```
!python {SCRIPTS_PATH + '/generate_tfrecord.py'} -x {IMAGE_PATH + '/train'} -l
{ANNOTATION_PATH + '/label_map.pbtxt'} -o {ANNOTATION_PATH + '/train.record'}
!python {SCRIPTS_PATH + '/generate_tfrecord.py'} -x{IMAGE_PATH + '/test'} -l
{ANNOTATION_PATH + '/label_map.pbtxt'} -o {ANNOTATION_PATH + '/test.record'}
```

Where SCRIPTS_PATH is the path where the generate_tfrecord.py is saved, IMAGE_PATH is the path for all images and labels from the dataset, and ANNOTATION_PATH is where you want to save the record files. The path for the label maps and record files are then inserted in the configuration file under the sections *train_input_reader* and *eval_input_reader* as shown in the script section bellow:

```
train_input_reader {
  label_map_path: "Tensorflow/workspace/annotations/label_map.pbtxt"
  tf_record_input_reader {
    input_path: "Tensorflow/workspace/annotations/train.record"
  }
}
eval_config {
  metrics_set: "coco_detection_metrics"
  use_moving_averages: false
}
eval_input_reader {
  label_map_path: "Tensorflow/workspace/annotations/label_map.pbtxt"
  shuffle: false
  num_epochs: 1
  tf_record_input_reader {
    input_path: "Tensorflow/workspace/annotations/test.record"
  }
}
```

Then, the path for the downloaded checkpoint is also passed to the configuration file, this will allow the model to be fine-tuned from the initial weights of this pre-trained model. The configuration file structure can change according with the downloaded pre-trained model, so the location of where the checkpoint path should be inserted can change. The easiest way is to look for the variable *fine_tune_checkpoint* under the *train_config* section as shown in the script section bellow.

```
}
  fine_tune_checkpoint_version: V2
  fine_tune_checkpoint: "Tensorflow/workspace/pre-trained-
models/faster_rcnn_resnet101_v1_640x640_coco17_tpu-8/checkpoint/ckpt-0"
  fine_tune_checkpoint_type: "detection"
  data_augmentation_options {
    random_horizontal_flip {
    }
  }

  max_number_of_boxes: 100
  unpad_groundtruth_tensors: false
  use_bfloat16: false  # works only on TPUs
}
```

The *fine_tune_checkpoint_type* is set to ‘detection’ and the variable *use_bfloat16* is set to ‘false’ since a GPU and a CPU will be used to the training instead of a Tensor processing unit (TPU).

Next, the number of classes and the image resolution are also changed, they are usually under the section with the name of the base architecture, e.g., ssd and faster_rcnn. the code section bellow shows the setting of the number of classes to 2, since the crucible task have the classes “Good” and “Bad”, and as mentioned previously, the image resolution is set to $320x320$, instead of the $640x640$ default setting of the pre-trained model.

```
model {
  faster_rcnn {
    num_classes: 2
    image_resizer {
      keep_aspect_ratio_resizer {
        min_dimension: 320
        max_dimension: 320
        pad_to_max_dimension: true
      }
    }
```

Finally, the batch size was also reduced from the default value of 64 from the downloaded pre-trained model. The new Batch size was set to 4 under the *train_config* section as shown in code section bellow.

```
train_config: {
  batch_size: 4
  sync_replicas: true
  startup_delay_steps: 0
  replicas_to_aggregate: 8
  num_steps: 25000
```

The resolution and batch size are set to low for calculation proposes, according to the calculation power of the available hardware. This will be better discussed in next section. For the same reason, the Faster R-CNN w/Inception Resnet V2 model was trained with 100 proposals instead of 300 initially showed in Table 1. The proposals setting can be seen in the following script under the *faster_rcnn* section.

```
    first_stage_nms_score_threshold: 0.0
    first_stage_nms_iou_threshold: 0.7
    first_stage_max_proposals: 100
    first_stage_localization_loss_weight: 2.0
    first_stage_objectness_loss_weight: 1.0
    initial_crop_size: 14
    maxpool_kernel_size: 2
    maxpool_stride: 2
```

All other advanced settings were left as default. The complete file configuration of all models can be seen in Appendices A-C.

Finally, the models are ready to be trained. For that, the script 'model_main_tf2.py' from the TensorFlow Object Detection API was used. An example of the command used for training one of the models can be seen bellow:

```
!python Tensorflow/models/research/object_detection/model_main_tf2.py --
model_dir=Tensorflow/workspace/models/my_faster_rcnn_inception_resnet_v2 --
pipeline_config_path=Tensorflow/workspace/models/my_faster_rcnn_inception_resn
et_v2/pipeline.config --num_train_steps=10000
```

Where 'model_dir' is the path where the new checkpoints will be saved and 'pipiline_config_path' is the path for the configuration file that was setup in the previous steps. Also, notice that the number of trained steps is also passed on the end of this command by the variable 'num_train_steps'.

4.5 TRAINING OF THE MODELS

The SSD with MobileNetV2 models were trained in a local computer using a NVIDIA GeForce GTX 960M GPU with 2GB memory, this GPU does not have the power computation and memory necessary to train the more complex models, Faster R-CNN with Resnet-101 and Faster R-CNN with Inception Resnet V2. For this reason, these two architectures were trained

in google Colab using Google's hosted computer which has a NVIDIA TESLA K80 GPU with 12GB memory.

Besides Google Colab having a very powerful GPU, the allocation for every user is limited if you use the free version. The most complex architecture Faster R-CNN with Inception Resnet V2 is the one that necessitates more resources and trying to train with a higher resolution, batch size or region proposals, the GPU runs out of resources. For this reason, a low resolution of 320x320 and a batch size 4 was set equally to all models, and for region proposals base models, the number of proposals was set to 100.

Because of this limitation on Google Colab, the Faster R-CNN with Inception ResNet V2 models were trained with only 4000 steps, more than that will exceed the limit section time on Google Colab and the user it's disconnected from the server. The SSD with MobileNetV2 models and Faster R-CNN with Resnet-101 models were trained with 10000 steps. The SSD with MobileNetV2 models took around 3h30min to train each one using the local GPU. The two Faster R-CNN w/Resnet-101 models took around 7h to train each using Google Colab. The training of the two Faster R-CNN w/Inception Resnet V2 models took around 10h each using Google Colab.

After all models were completed trained, their corresponding final checkpoints were saved and stored to be used for evaluation.

5 EXPERIMENTAL SETUP

In this chapter, the equipment used for the test of the models will be described, as well as the developed software used to perform the crucible position predictions.

5.1 USED EQUIPMENT

For the selection of the camera to be used with the CNN models, the following criteria were used: Resolution, Sensor type, Shutter, and image type.

The required resolution is determined by the camera's desired field of view (FOV) and the size of the smallest feature to be detected. A minimum of two pixels per smallest feature was used to ensure accurate measurement on the image.

For the crucible position detection, the area of view was taken as $800\ mm\ x\ 600\ mm$ and the smallest feature was taken as $1\ mm$ (the width of the crucible axis).

$$Image\ Resolution = 2\left(\frac{Field\ of\ View\ (FOV)}{Snallest\ Feature}\right) = 2\left(\frac{800\ mm}{1\ mm}\right) = 1600$$

The Minimum Resolution needed is around $1600\ x\ 1600\ pixels$ or $2.5MP$.

The two most common types of electronic image sensors are charge-coupled devices (CCD) and active-pixel sensors (CMOS). A CCD sensor's pixel values can only be accessed on a per-row basis. One by one, each row of pixels is moved into a readout register. A CMOS sensor, on the other hand, enables for independent reading of each pixel.

The CMOS sensor is preferable because it is less expensive and uses less energy without compromising visual quality. It can also achieve higher frame rates because of the simultaneous readout of pixel data. CCD is better suited to applications that require extremely low-noise images, such as astronomy.

A rolling shutter exposes the pixel rows in a specific order, such as top to bottom, whereas a global shutter exposes each pixel to incoming light at the same moment. The major advantage of the global shutter over the rolling shutter is that it does not suffer from the same distortion problems. The global shutter has the disadvantage of being more expansive; however, because the crucible position classification task will not be performed at high frame rates but rather in a single frame, a camera with a rolling shutter is sufficient.

A monochrome camera should be considered for specialized applications where color information isn't necessary. While having a smaller sensor size and lower sensor resolution, it will boost light sensitivity and spatial resolution. As mentioned in Chapter 3 The Dataset, For the crucible position detection, the color of the Crucible does not change and the color of the crucible support or the crucible filler does not affect the classification, the Monochrome is the most suitable for this application.

The camera selected for testing of the models was the Allied Vision Alvium 1800 U-500 industrial camera [38] together with the TECHSPEC 4mm UC Series Fixed Focal Length Lens [39]. This industrial camera covers all the above specifications and also has its own software

and python library: Vimba [40]. The Vimba viewer software can be used to adjustment of the camera settings, e.g., exposure, gain, black level, etc. These setting can be stored in an XML file and then loaded to the camera at any time using the Vimba library for python. This same library will be used to the acquisition of images frames fed to the detection models. Figure 46 shows the industrial camera mounting inside the VITRIOX® ELECTRIC machine for the testing of the models.

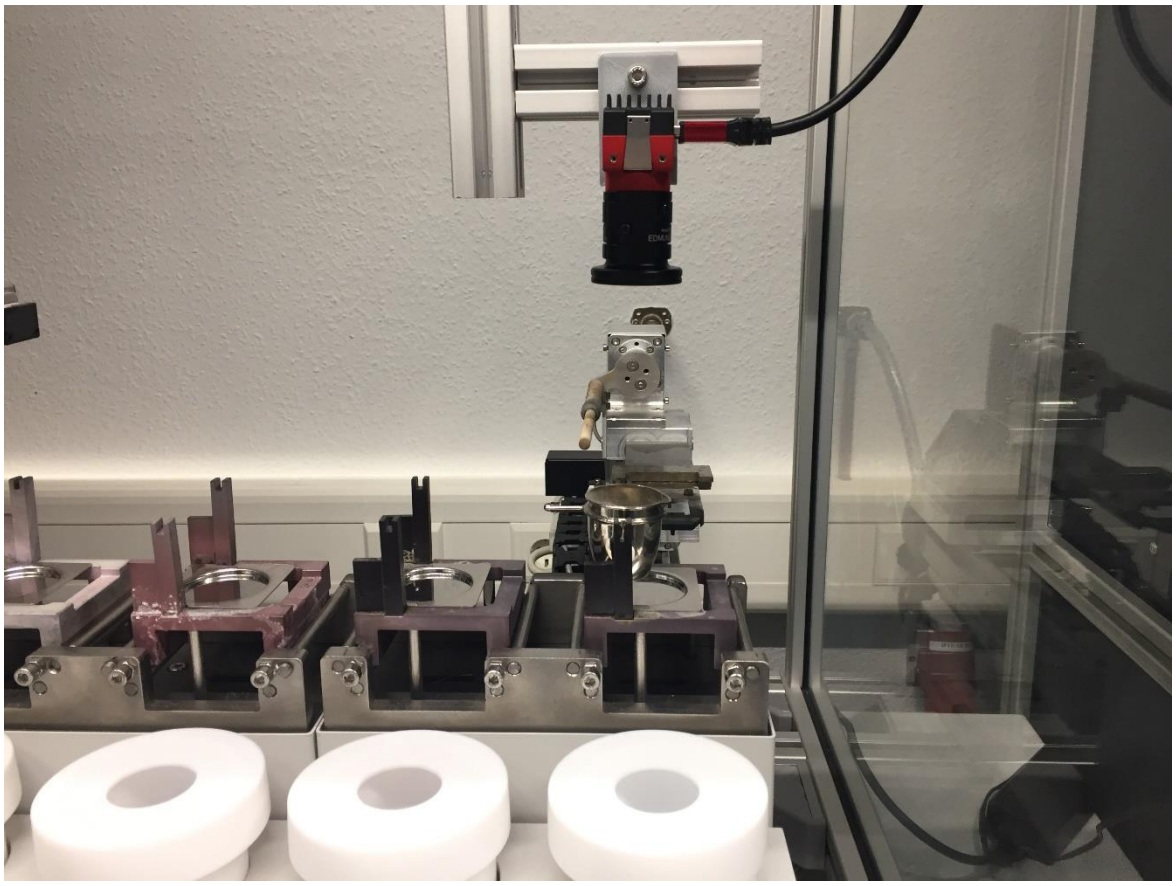


Figure 46: Alvium 1800 U-500 camera mounting inside VITRIOX® ELECTRIC machine.

5.2 THE FX_CRUCIBLEPOSITIONCHECKER SOFTWARE

A custom software for the models evaluation was created using Python 3 and Tkinter [41]. Tkinter is a python library to build GUI (Graphic User Interface) applications. The software developed was called *FX_CruciblePositionChecker* and was installed together with the industrial camera on the computer mounted in the VITRIOX® ELECTRIC machine.

The FX_CruciblePositionChecker software has two buttons: check and reset. The check button gets an image frame from the camera and passes through the loaded detection model. As shown in Figure 47 and Figure 48, if the output of the prediction is “Good” for the handle position model or for both axis for the axis position model, a green signal is given. If one of the outputs is “Bad” a red signal is shown. For visualization of the results, the detection regions and the corresponding prediction confidence are drawn in the current image frame. The reset button resets the camera view for a new prediction.

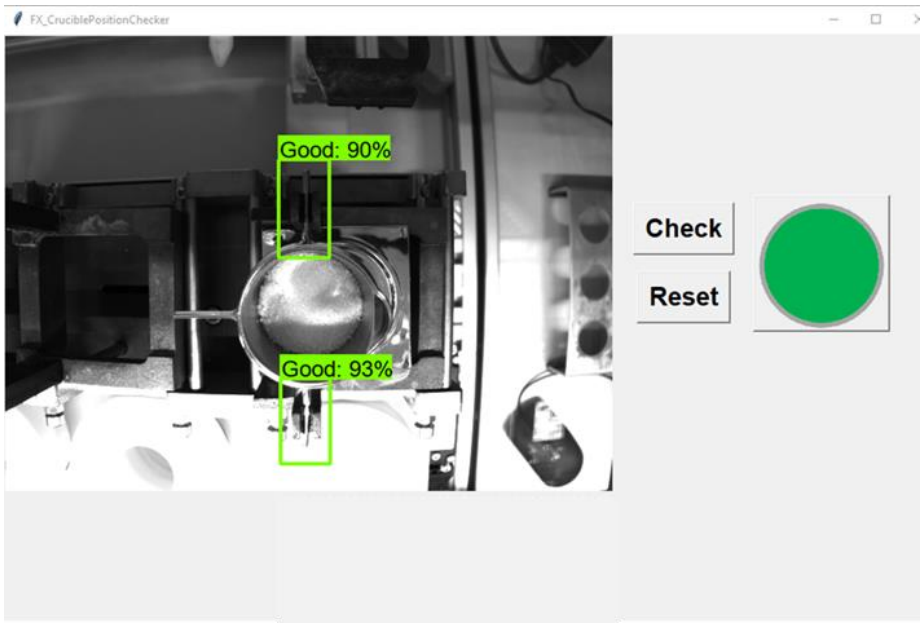


Figure 47: Good axis position detection.

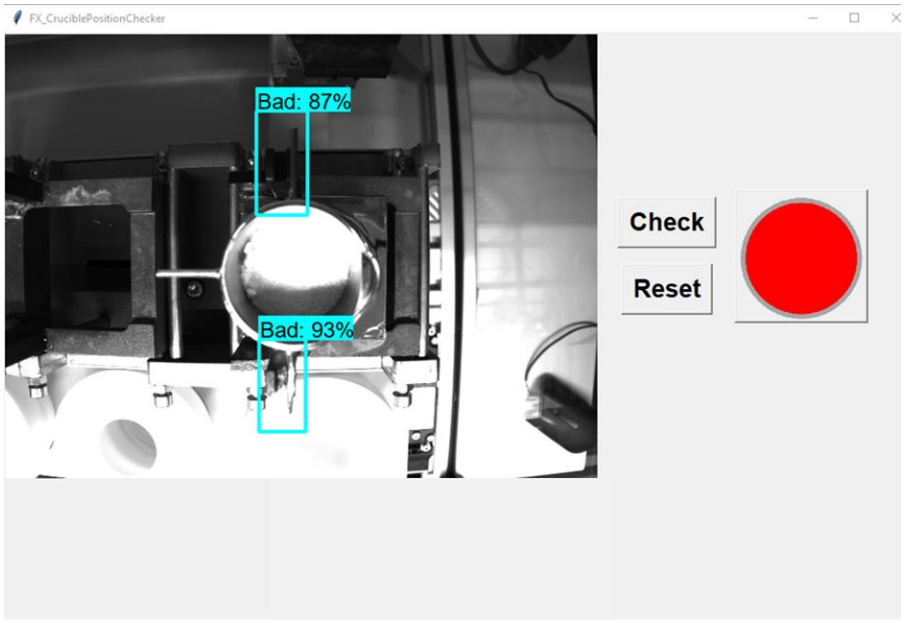


Figure 48: Bad axis position detection.

The main parts of the *FX_CruciblePositionChecker* software will be explained bellow, for the complete code, please refer to Appendix D. First, the path of all the saved trained models that will be used needs to be defined.

```
ANNOTATION_PATH = 'annotations' #Name of the folder where the label and record
files are saved
MODEL_PATH = 'models' #Name of the folder where the models are saved
CONFIG_PATH_AXIS = MODEL_PATH+'/my_faster_rcnn_resnet101_v1/pipeline.config'
#path of the configuration file of the axis model
CONFIG_PATH_HANDLE =
MODEL_PATH+'/my_faster_rcnn_inception_resnet_v2/pipeline.config' #path of the
configuration file of the handle model
```

```
CHECKPOINT_PATH_AXIS = MODEL_PATH+'/my_faster_rcnn_resnet101_v1/' #Name of the
folder where the axis model's checkpoints are saved
CHECKPOINT_PATH_HANDLE = MODEL_PATH+'/my_faster_rcnn_inception_resnet_v2/'
#Name of the folder where the handle model's checkpoints are saved
```

The next step is to import all necessary dependencies. All the libraries used in the application are described below.

```
import tensorflow as tf #imports tensorflow library

#Imports Google's object detection API
from object_detection.utils import config_util
from object_detection.utils import label_map_util
from object_detection.utils import visualization_utils as viz_utils
from object_detection.builders import model_builder

import cv2 #Imports OpenCV for image data handling
#Import libraries that handle system files and variables
import os
import sys

from typing import Optional #Import typing libraries
import numpy as np #Imports library for handle array manipulation
#Imports the industrial camera library
from vimba import*
from vimba.c_binding.vimba_common import VmbPixelFormat
#Import Tkinter libraries for GUI creation
import tkinter as tk
from tkinter import font as tkFont
from PIL import ImageTk, Image
```

Then, the application asks the user which crucible classification type needs to be performed. Where ‘1’ is for the axis position classification and ‘2’ is for the handle position classification, this value is stored in a variable so that it can be used later.

```
classification_type = int(input("Enter which position classification task need
to be performed (1=axis;2=handle): "))
```

According to the selected classification task, the right trained model configuration file is loaded to the application.

```
if classification_type == 1:
    configs = config_util.get_configs_from_pipeline_file(CONFIG_PATH_AXIS)
elif classification_type == 2:
    configs = config_util.get_configs_from_pipeline_file(CONFIG_PATH_HANDLE)
```

```
detection_model = model_builder.build(model_config=configs['model'],
is_training=False)
```

In the same way, the final checkpoints from the trained CNN models are loaded according with the classification task type.

```
ckpt = tf.compat.v2.train.Checkpoint(model=detection_model)
if classification_type == 1:
    ckpt.restore(os.path.join(CHECKPOINT_PATH_AXIS, 'ckpt-
11')).expect_partial()
elif classification_type == 2:
    ckpt.restore(os.path.join(CHECKPOINT_PATH_HANDLE, 'ckpt-
5')).expect_partial()
```

Next, the function to actually perform the crucible position classification predictions is defined. This function receives an image as input parameter and returns a set of detections as objects.

```
@tf.function
def detect_fn(image):
    image, shapes = detection_model.preprocess(image)
    prediction_dict = detection_model.predict(image, shapes)
    detections = detection_model.postprocess(prediction_dict, shapes)
    return detections
```

For the application to be able to know which classification means, the label map is also loaded.

```
category_index =
label_map_util.create_category_index_from_labelmap(ANNOTATION_PATH+'/label_map
.pbtxt')
```

To differentiate to when the application is performing a prediction from when it's only on display mode, two variables were created: *display_option*, that indicates the current application mode ('1' for only display mode and '2' for prediction mode), and *predict*, who determines if a prediction needs to be made.

```
display_option = "1" #Set the display option of the frame (1=current camera
frame;2=predicted result with drawn bounding boxes)
predict = False #Flag to indicate if the software should make a prediction of
the current frame of not
```

As mentioned in the previous section, the Alvium 1800 U-500 industrial camera settings are stored in an XML document. Below is the function that load these custom settings from the *settings.xml* file to the camera.

```
    with Vimba.get_instance():
        with get_camera(cam_id) as cam:

            # Load camera settings from file.
            settings_file = 'settings.xml'
            cam.load_settings(settings_file, PersistType.All)
```

Next, the application window, with the buttons and labels is created. Below is the code for the creation of the main window and frames where the buttons and labels will be placed. The main text font used in the application is also set.

```
    #create th emain window of the appliation
    root = tk.Tk()
    root.title("FX_CruciblePositionChecker")


    root.geometry('{}x{}'.format(1000, 800)) #set the application window size
    #create the main frame of the application
    main_frame = tk.Frame(root)
    main_frame.pack(fill=tk.BOTH)
    #cretae the label where the camera frames will be displayed
    lmain = tk.Label(main_frame)
    lmain.grid(row=0,column=0)
    #create the inner frame where the buttons will be inserted
    inner_frame = tk.Frame(main_frame)
    inner_frame.grid(row=0,column=1)

    #define the text font for the buttons
    mainFont = tkFont.Font(family="Arial", size=20, weight="bold")
```

The code for the creation of the predict and reset buttons is showed next. The *PredictBtnClick()* function set the necessary flags, so that the application is set to prediction mode. While the ResetBtnClick() set the application mode to only display.

```
#predict button function to set the application mode to prediction
    def PredictBtnClick():
       global display_option,predict
       display_option = "2"
       predict = True

    #Create the predict button and assign to its function
    PredictBtn =
tk.Button(inner_frame,text="Check",command=PredictBtnClick,font=mainFont)
    PredictBtn.grid(row=0,column=0,padx=(20,0))

    #Reset button function to set the application mode to normal view
    def ResetBtnClick():
```

```
        global display_option
        display_option = "1"

    #Create the reset button and assign to its function
    ResetBtn
=  tk.Button(inner_frame,text="Reset",command=ResetBtnClick,font=mainFont)
    ResetBtn.grid(row=1,column=0,padx=(20,0))
```

Next, the creation of the status button is shown. The status button will display “green” or “red” according to the prediction results, as seen in Figure 47 and Figure 48.

```
    #Get image of "empty button" to be displayed when there is no prediction
displayed
    StatussBtnImagepath = ImagesPath + "\empty button.png"
    StatussBtnImage =
ImageTk.PhotoImage(Image.open(StatussBtnImagepath).resize((140, 140),
Image.ANTIALIAS))
    #Create the button that will display the status of the picture
frame('empty'= no prediction are been displayed; ‘green'= the model predicted
as 'Good’; ‘red'= the model predicted as 'Bad')
    StatussBtn = tk.Button(inner_frame,width=140,height=140)
    StatussBtn.grid(row=0,column=1,padx=(20,0),rowspan=2)

    #Get image of "Green button" to be displayed when the model is showing a
good position prediction
    GreenBtnImagepath = ImagesPath + "\green button.png"
    GreenBtnImage =
ImageTk.PhotoImage(Image.open(GreenBtnImagepath).resize((140, 140),
Image.ANTIALIAS))
    #Get image of "Red button" to be displayed when the model is showing a bad
position prediction
    RedBtnImagepath = ImagesPath + "\\red button.png"
    RedBtnImage = ImageTk.PhotoImage(Image.open(RedBtnImagepath).resize((140,
140), Image.ANTIALIAS))
```

The main function in the application is called *show_frame()*, it’s the function that will be running in loop. It’s the function that actually display the current image to the user, as well as checking if a prediction needs to be made and actually calls the detection function. This is a big function, only the main parts are showed below. For more details, again, refer to the complete code in Appendix D. The Vimba library is used to get the current image frame and store it in the variable *frame*.

```
with Vimba.get_instance():
            with get_camera(cam_id) as cam:

                # Acquire single frame synchronously
                frame = cam.get_frame ()
```

If the application is on the display mode ‘1’, the application simply shows the frame to the user through the label *lmain*.

```
 #If the display flag is set to 1, the application shows the current frame of
the camera without any predictions
                    if display_option == "1":
                        try:
                            image_np = frame.as_numpy_ndarray()
                            cv2image = cv2.resize(image_np, (648, 486))
                            img = Image.fromarray(cv2image)
                            imgtk = ImageTk.PhotoImage(image=img)
                            lmain.imgtk = imgtk
                            lmain.configure(image=imgtk)
                            StatussBtn.config(image=StatussBtnImage)
                        except Exception as err:
                            print(err)
```

If the application is on the prediction mode ‘2’, the application first calls the detection function defined previously and store the resulting prediction objects in a variable.

```
  detections = detect_fn(input_tensor)
  num_detections = int(detections.pop('num_detections'))
  detections = {key: value[0, :num_detections].numpy()
         for key, value in detections.items()}
  detections['num_detections'] = num_detections
  # detection_classes should be ints.
  detections['detection_classes'] =
detections['detection_classes'].astype(np.int64)
```

With these resulting predictions, the application uses the TensorFlow Object Detection API function *visualize_boxes_and_labels_on_image_array()* to drawn the detection results into the image that will be displayed to the user.

```
#function to drawn the predicted boxes in the current image
viz_utils.visualize_boxes_and_labels_on_image_array(
   image_np_with_detections,
   detections['detection_boxes'],
   detections['detection_classes']+label_id_offset,
   detections['detection_scores'],
   category_index,
   use_normalized_coordinates=True,
   max_boxes_to_draw=5,
   min_score_thresh=.3,
   agnostic_mode=False)
```

The *max_boxes_to_draw* value sets the maximum number of detection boxes that will be showed. While the *min_score_thresh* sets the minimum confidence score that a prediction must have to be considered and displayed.

Finally, according with results, the status button is changed to “green” or “red”. In case of the axis position classification, both axis needs to present a ‘Good’ detection to set the status button to “green”, otherwise the button is set to “red”. For the handle position classification, only the handle prediction is taken into consideration.

```
  #If both axes were detected as a good position, the status button is set to
green, anything else, the status button is set to red
  global RedBtnImage,GreenBtnImage
  if category_index[detections['detection_classes'][0] +
label_id_offset]['name'] == "Good" and
category_index[detections['detection_classes'][1] + label_id_offset]['name']
== "Good" and classification_type == 1:
      StatussBtn.config(image=GreenBtnImage)
  Elif category_index[detections['detection_classes'][0] +
label_id_offset]['name'] == "Good" and classification_type == 2:
      StatussBtn.config(image=GreenBtnImage)
  else:
      StatussBtn.config(image=RedBtnImage)
```

All trained models were then tested using the *FX_CruciblePositionChecker* software and evaluated according to the resulting predictions and confidence score.

6 EVALUATION AND RESULTS

In this chapter, the three models for both handle and axis position detection will be evaluated according to their performance on the crucible dataset. The used method for evaluation will be also explained in the next section.

6.1 METHOD USED FOR EVALUATION

For evaluation of the models, a *confusion matrix* will be used, the template used in this thesis was created following the guideline presented by Saito and Rehmsmeier [42]. It's a well-known technique for describing the performance of a classification algorithm. The confusion matrix is designed for binary-class classification by default, but it can be extended to handle multi-class classification. In Table 3, the model of the confusion matrix for the crucible classification task is presented.

Table 3: Crucible classification task confusion matrix.

Crucible Classification		Predicted Class	
		Good	Bad
True Class	Good	D_{GG}	D_{GB}
	Bad	D_{BG}	D_{BB}

Where D_{GG} is the number of a good crucible position that were correctly detected as "Good" and also called as *True Positive*. D_{GB} is the number of good crucible position wrongly detected as "Bad", also called *False Negative*. D_{BG} indicates the number of bad crucible positions that were wrongly detect as "Good", also called *False Positive*. And finally, D_{BB} is the number of bad crucible positions that were correctly detected as "Bad", also called *True Negative*.

Various evaluation metrics can be taken from the confusion matrix, for this thesis, two main ones will be used: accuracy (ACC) and error rate (ERR). The number of right predictions divided by the total number of guesses in the dataset delivers the accuracy. The highest level of accuracy is 1.0, while the lowest level is 0.0. The formula to calculate the accuracy is shown in Eq. 17.

$$ACC = \frac{D_{GG} + D_{BB}}{N}$$

Eq. 17

Where N is the number of totals predictions made. To calculate the error rate, the number of wrong predictions is divided by total number of predictions. Opposite to the accuracy number ACC, with an ERR of 0.0 the model has the best error rate while if the model has an ERR of 1.0, it represents the worst error rate. The formula to calculate the error rate is shown in Eq. 18.

$$ERR = \frac{D_{GB} + D_{BG}}{N}$$

Eq. 18

Another way to calculate the error rate is by the accuracy value ACC. Eq. 19 demonstrate that.

$$ERR = 1 - ACC = 1 - \frac{D_{GG} + D_{BB}}{N}$$

Eq. 19

With the accuracy and error rate, the three chosen CNN architectures will be compared for the two tasks: Axis position classification and handle position classification.

Taking into consideration that the VITRIOX® ELECTRIC machine will be placed across different laboratories in different locations, each with different ambient light sources, the models will also be compared against their ability to perform well in variable picture light conditions.

For comparing the robustness of the models according with the ambient light, each model was tested in three different exposures. Figure 49 shows the image comparison of these three exposure times.

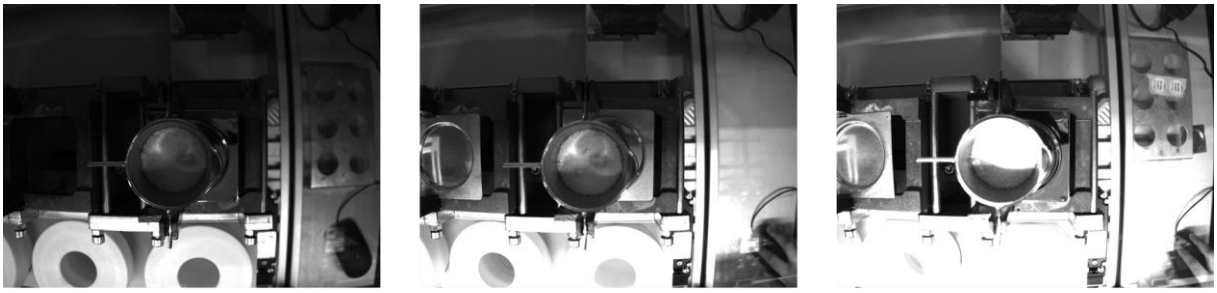


Figure 49: Crucible pictures taken with different exposure times.

For each exposure, 100 tests were made for each only axis model and 50 tests were made for each only handle model, adding 1350 tests in total. The following sections presents the results of all six trained modules.

6.2 ONLY AXIS MODELS

First, the results for each axis model will be presented and discussed. The results of each model were documented in the form of a excel sheet. The Appendices E-M show the complete results for the three only axis models for each exposure time.

6.2.1 EVALUATION METRICS

As mentioned in section 4.1 Important Software Components, TensorBoard will be used to analyse the training of each model. During training, the objective of the model is to reduce the loss function [Eq. 3], so comparing the loss function and how it's decreased during training can already give an idea of the model's performance. Figure 50 shows the total loss function for SSD with MobileNetV2 model for the crucible axis position classification in respect to the number of training steps.

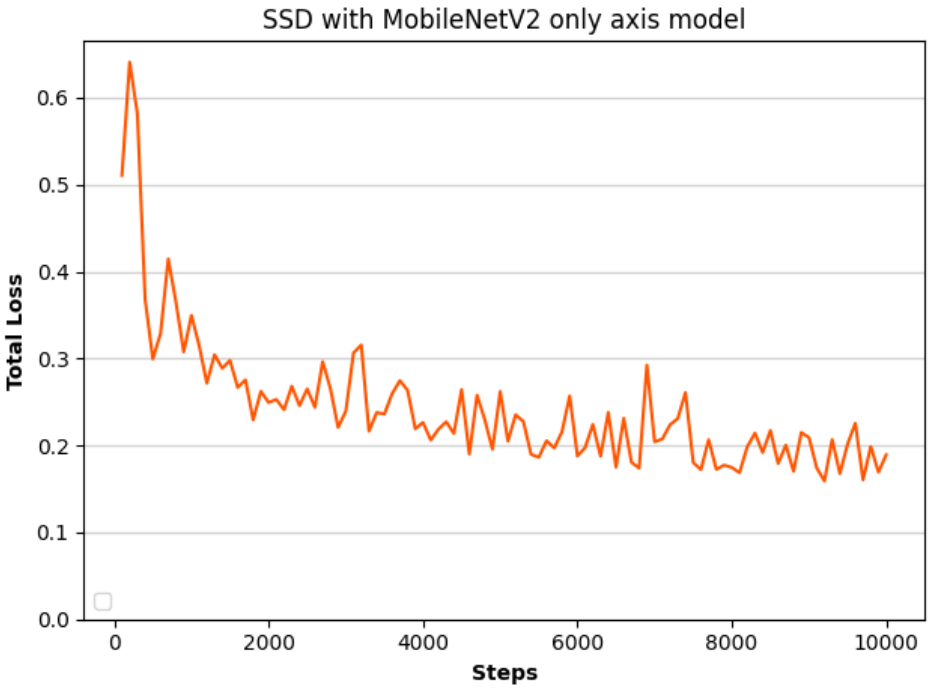


Figure 50: SSD with MobileNet model for axis position classification total loss.

The total loss by each training step for the Faster R-CNN with Resnet-101 model for axis position classification is shown in Figure 51.

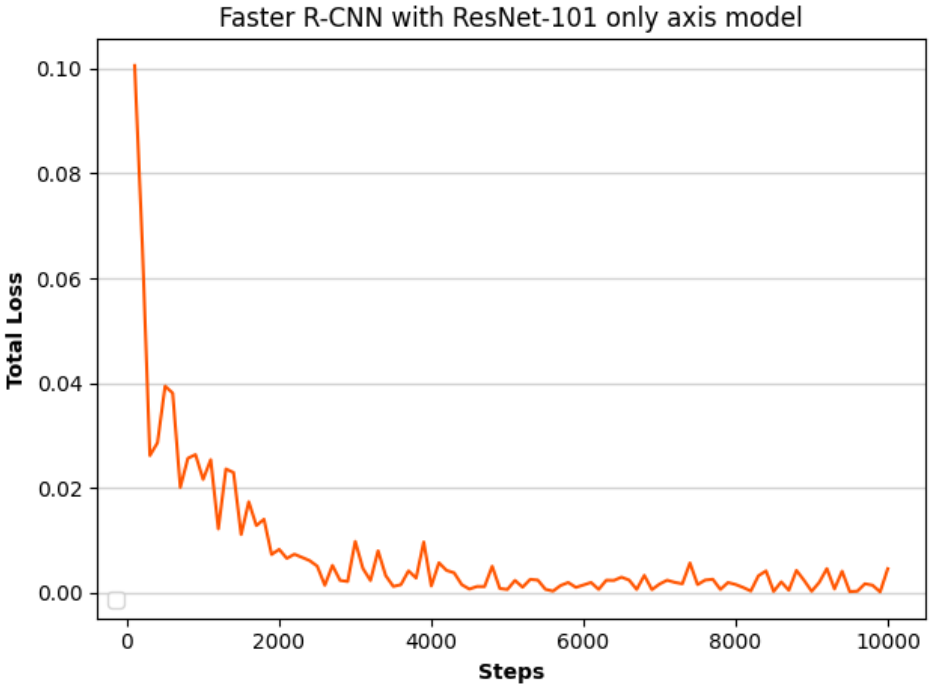


Figure 51: Faster R-CNN with ResNet-101 model for axis position classification total loss.

Finally, the total loss by the training steps for the Faster R-CNN with Inception ResNet V2 model for axis position classification is shown in Figure 52.

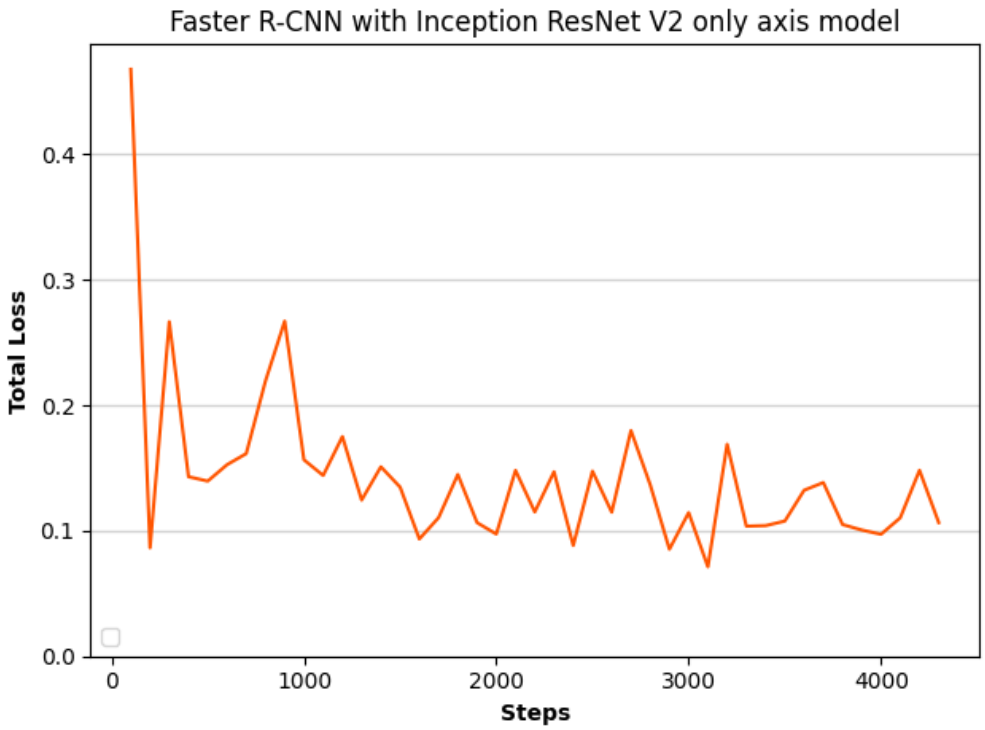


Figure 52: Faster R-CNN with Inception ResNet V2 model for axis position classification total loss.

Just by looking the total loss decreasing, it can be seen that the Faster R-CNN with Resnet-101 model is the one with a smaller total loss and quicker convergence, thus, it's to be assumed it will be the one with better results. But keep in mind that a model can present low loss function value but still show bad results, one reason for that it's overfitting as explained in section 2.1.5 Under- and Overfitting.

Since the axis position classification task have four different bad axis positions and one good axis position, the only axis models were test 100 times each, 50 images with a good position and 50 with a bad position. The confusion matrix for the SSD with MobileNetV2 model results is presented in Table 4.

Table 4: SSD with MobileNet model for axis position classification confusion matrix.

SSD with MobileNetV2 model			Predicted Class	
			Good	Bad
$Exposure = 10002{,}435\,\mu s$	True Class	Good	3	47
		Bad	0	50
$Exposure = 21857{,}652\,\mu s$	True Class	Good	43	7
		Bad	0	50
$Exposure = 32002{,}609\,\mu s$	True Class	Good	50	0
		Bad	0	50

Table 5 shows the confusion matrix for the Faster R-CNN with ResNet-101 model.

Table 5: Faster R-CNN with ResNet-101 model for axis position confusion matrix.

Faster R-CNN with ResNet-101 model			Predicted Class	
			Good	Bad
$Exposure = 10002{,}435\,\mu s$	True Class	Good	50	0
		Bad	2	48
$Exposure = 21857{,}652\,\mu s$	True Class	Good	48	2
		Bad	6	44
$Exposure = 32002{,}609\,\mu s$	True Class	Good	41	9
		Bad	0	50

And finally, Table 6 shows the confusion matrix for the Faster R-CNN with Inception ResNet V2 model.

Table 6: Faster R-CNN with Inception ResNet V2 model for axis position confusion matrix.

Faster R-CNN with Inception ResNet V2 model			Predicted Class	
			Good	Bad
$Exposure = 10002{,}435\,\mu s$	True Class	Good	36	14
		Bad	1	49
$Exposure = 21857{,}652\,\mu s$	True Class	Good	25	25
		Bad	0	50
$Exposure = 32002{,}609\,\mu s$	True Class	Good	27	23
		Bad	0	50

From these confusion matrices and the total number of predictions $N = 100$, the accuracy and error for each model can be calculated. The calculated accuracy and error for each model is presented in Table 7.

Table 7: Accuracy and error for the only axis position models.

		SSD with MobileNetV2	Faster R-CNN with ResNet-101	Faster R-CNN with Inception ResNet V2
$Exposure = 10002{,}435\,\mu s$	Accuracy [%]	53	98	85
	Error [%]	47	2	15
$Exposure = 21857{,}652\,\mu s$	Accuracy [%]	93	92	75
	Error [%]	7	8	25
$Exposure = 32002{,}609\,\mu s$	Accuracy [%]	100	91	77
	Error [%]	0	9	23

6.2.2 DISCUSSION OF RESULTS

For a better view of the results, the accuracy for each model and exposure time was put it in a graph as seen in Figure 53.

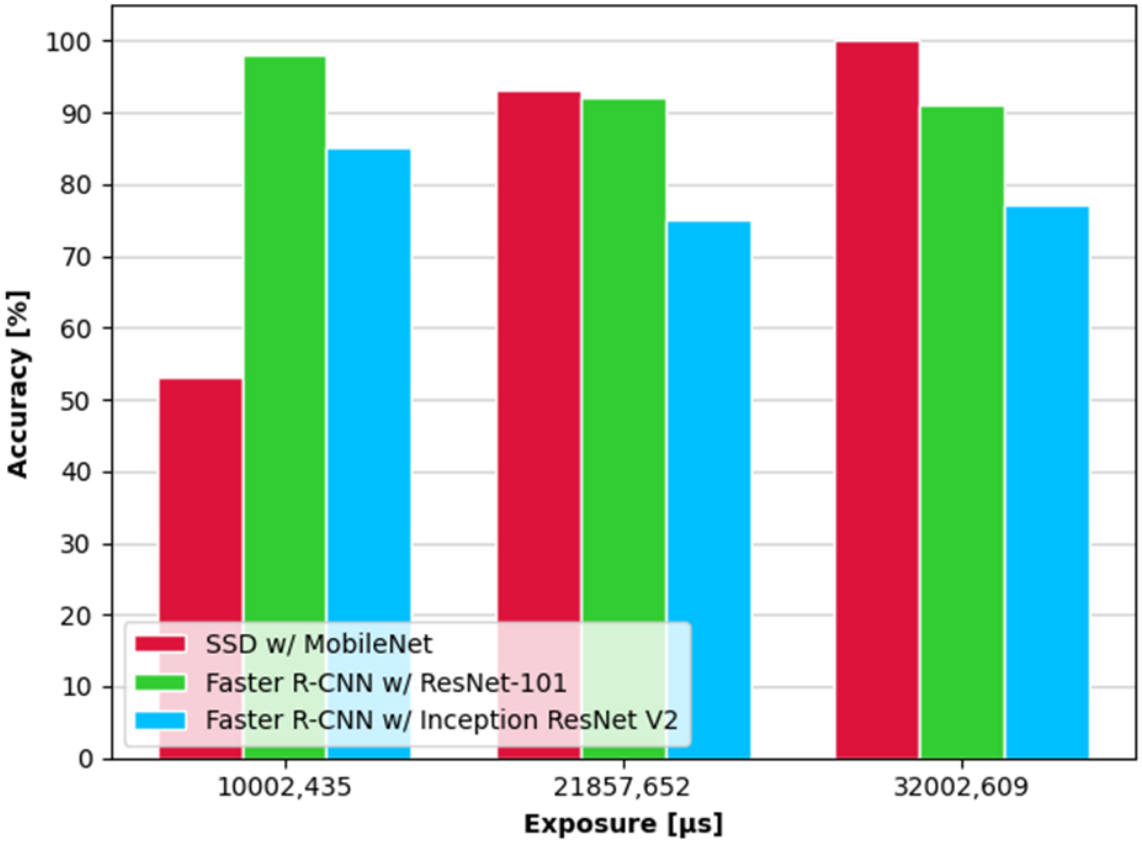


Figure 53: Accuracy for each only axis model by exposure time.

Looking at the graph, it's easy to see that besides the SSD with MobileNetV2 model having a good result for a bright lighting, the accuracy decreases a lot for a darker ambient light, showing a variance of 47% between the higher and lower exposure time. The two Faster R-CNN models show a more stable set of predictions for different lightning.

As assumed in the previous section, the Faster R-CNN with ResNet-101 model does shows the better overall results. With accuracy above 90% for all three exposure times and with a variance of less than 10% in the predictions when light goes a little darker or brighter.

Besides the Faster R-CNN with Inception ResNet V2 model also having a good balance of accuracy values between the different exposure times, it still shows a lower accuracy, with a value range between 75% – 85%, than the Faster R-CNN with ResNet-101 model.

6.3 ONLY HANDLE MODELS

As the only axis models, the results for the only handle models for each architecture and exposure time are documented in excel sheets shown in Appendices N-V.

6.3.1 EVALUATION METRICS

First, the total loss function for SSD with MobileNet model for the crucible handle position classification in respect to the number of steps is shown in Figure 54.

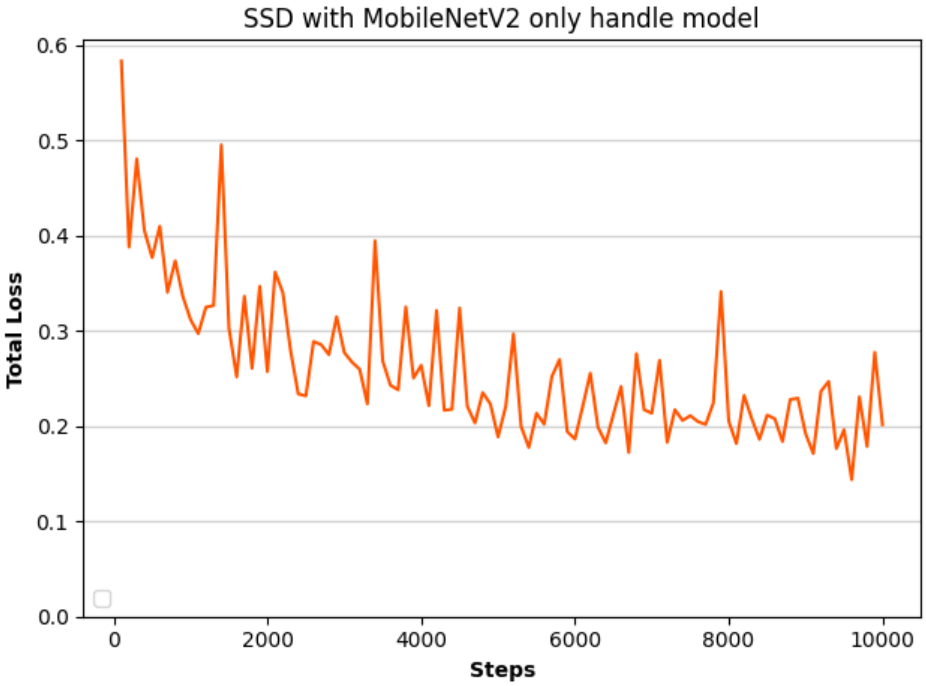


Figure 54: SSD with MobileNet model for handle position classification total loss.

The total loss by each training step for the Faster R-CNN with Resnet-101 model for handle position classification is shown in Figure 55.

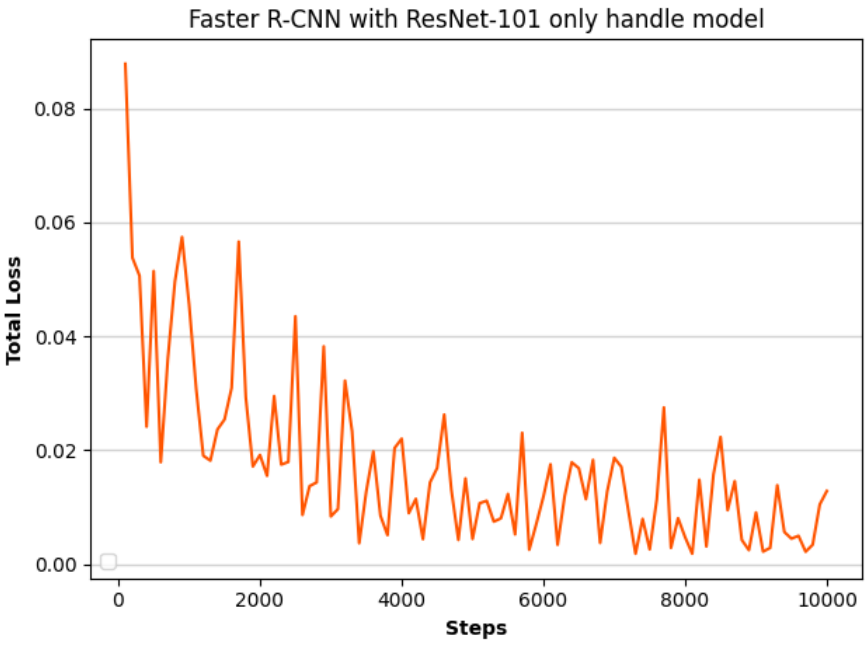


Figure 55: Faster R-CNN with ResNet-101 model for handle position classification total loss.

Finally, the total loss by the training steps for the Faster R-CNN with Inception ResNet V2 model for handle position classification is shown in Figure 56.

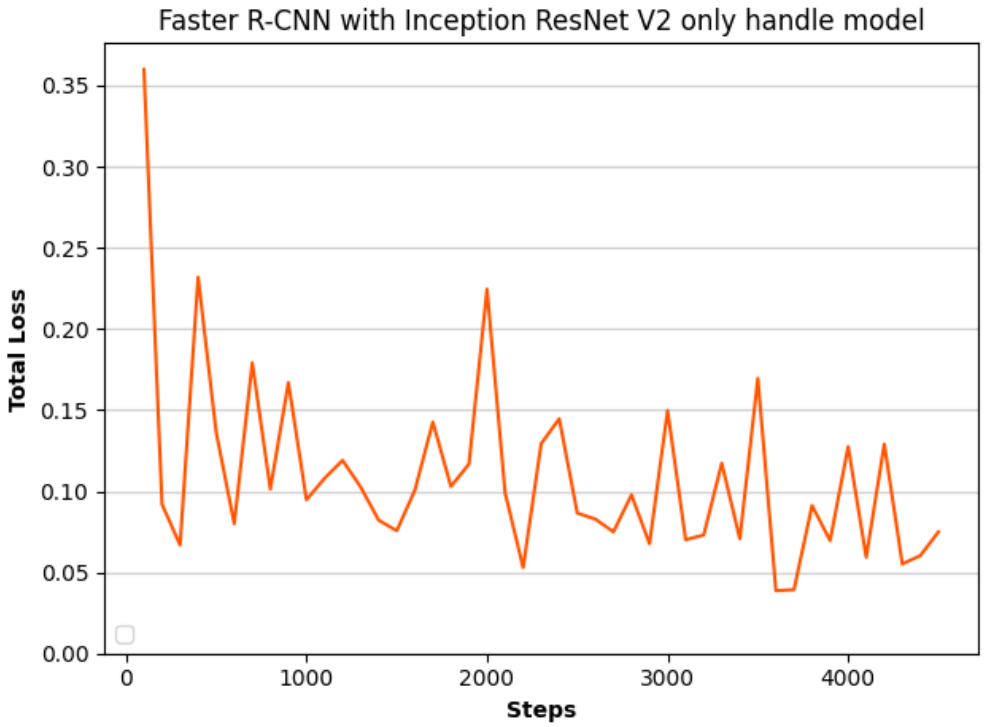


Figure 56: Faster R-CNN with Inception ResNet V2 model for handle position classification total loss.

For the only handle models, the training for the Faster R-CNN with Inception ResNet V2 model is the one that presents the smallest final loss value.

Since the handle position classification task, only has one bad handle position and one good handle position, the only handle models were test 50 times each, 25 images with a good position and 25 with a bad position. The confusion matrix for the SSD with MobileNet model results is present in Table 8.

Table 8: SSD with MobileNet model for handle position classification confusion matrix.

SSD with MobileNet model			Predicted Class	
			Good	Bad
$Exposure = 10002{,}435\,\mu s$	True Class	Good	21	4
		Bad	23	2
$Exposure = 21857{,}652\,\mu s$	True Class	Good	25	0
		Bad	22	3
$Exposure = 32002{,}609\,\mu s$	True Class	Good	25	0
		Bad	22	3

Table 9 shows the confusion matrix for the Faster R-CNN with ResNet-101 model.

Table 9: Faster R-CNN with ResNet-101 model for handle position confusion matrix.

Faster R-CNN with ResNet-101 model			Predicted Class	
			Good	Bad
$Exposure = 10002{,}435\,\mu s$	True Class	Good	25	0
		Bad	4	21
$Exposure = 21857{,}652\,\mu s$	True Class	Good	25	0
		Bad	0	25
$Exposure = 32002{,}609\,\mu s$	True Class	Good	25	0
		Bad	7	18

And finally, Table 10 shows the confusion matrix for the Faster R-CNN with Inception ResNet V2 model.

Table 10: Faster R-CNN with Inception ResNet V2 model for handle position confusion matrix.

Faster R-CNN with Inception ResNet V2 model			Predicted Class	
			Good	Bad
$Exposure = 10002{,}435\,\mu s$	True Class	Good	25	0
		Bad	0	25
$Exposure = 21857{,}652\,\mu s$	True Class	Good	25	0
		Bad	0	25
$Exposure = 32002{,}609\,\mu s$	True Class	Good	25	0
		Bad	0	25

From these confusion matrices and the total number of predictions $N = 50$, the accuracy and error for each model can be calculated. The calculated accuracy and error for each model is presented in Table 11.

Table 11: Accuracy and error for the only handle position models.

		SSD with MobileNet	Faster R-CNN with ResNet-101	Faster R-CNN with Inception ResNet V2
$Exposure = 10002{,}435\,\mu s$	Accuracy [%]	46	92	100
	Error [%]	54	8	0
$Exposure = 21857{,}652\,\mu s$	Accuracy [%]	56	100	100
	Error [%]	44	0	0
$Exposure = 32002{,}609\,\mu s$	Accuracy [%]	56	86	100
	Error [%]	44	14	0

6.3.2 DISCUSSION OF RESULTS

As done for the only axis models, the accuracy for each only handle model and exposure time was put it in a graph shown in Figure 57.

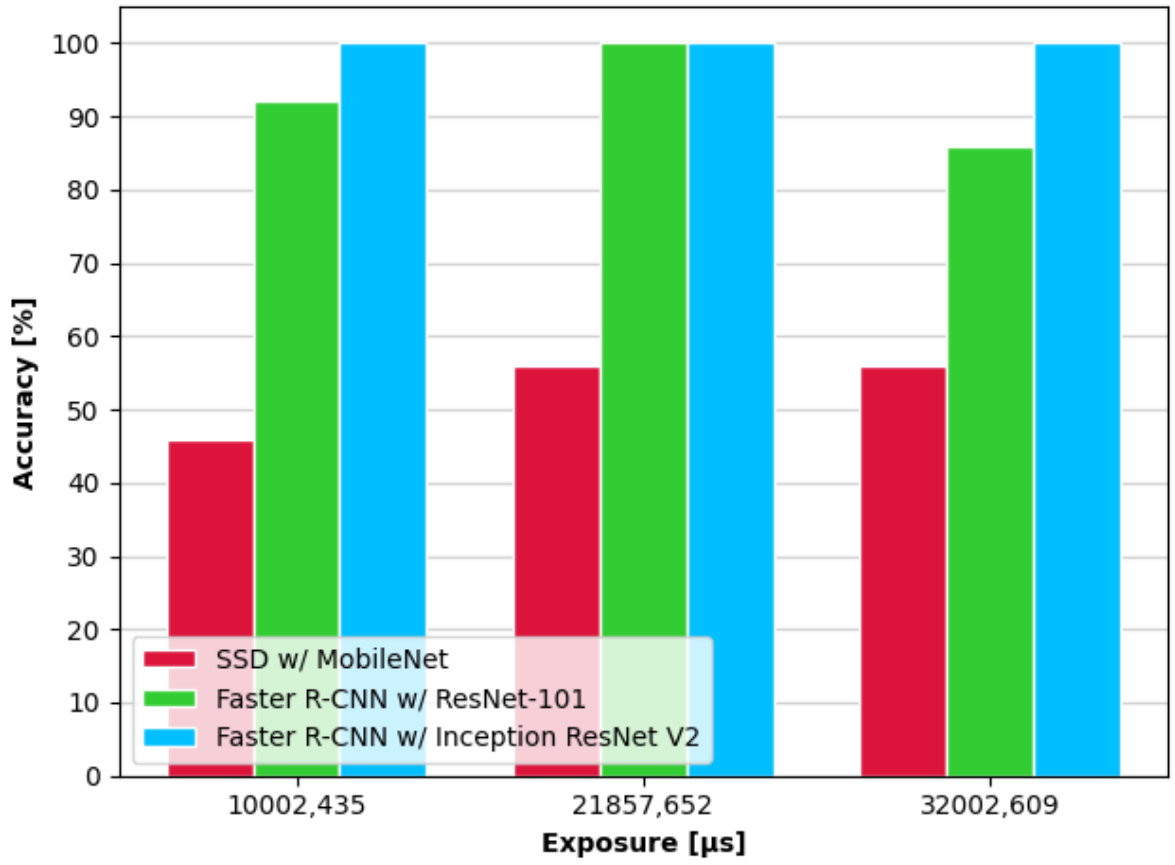


Figure 57: Accuracy for each only handle model by exposure time.

For the handle position, the SSD with MobileNetV2 model shows the lowest accuracy for all three exposure times, with a value range between $46\% - 56\%$. The Faster R-CNN with ResNet-101 model again showed good results, but with a higher value variance of 14% between the lower and higher exposure time, and a slight lower accuracy range of $86\% - 100\%$.

The Faster R-CNN with Inception ResNet V2 model shows the better results overall for the handle position classification task for all three exposure times, since it classified all positions correctly and had an accuracy of 100% for all three exposure times.

6.4 RESULTS SUMMARY

In this chapter, the models were evaluated based on their accuracy and error values, as well as their robustness against disturbances on the picture's light. The Faster R-CNN with ResNet-101 model showed to be the best choice for the crucible axis position classification task. Since is the one with high accuracy values while being the most robust against disturbances in the picture lighting.

The best choice for the crucible handle position classification task is the Faster R-CNN with Inception ResNet V2 model. Since it presented an equal 100% accuracy for all exposure times, showing to be the most robust model against disturbances in the image lightning.

7 CONCLUSION AND FUTURE WORK

In the previous chapter, the evaluation, and results of the three chosen Convolutional Neural Network were presented. In the first section of this chapter, the thesis methodology and findings are summarized and discussed in comparison with the objectives stated in the introduction of this thesis and the initial expectations based on related work as well as the limitations found during the training process. Later on, in the second section, suggestions for future work are provided.

7.1 CONCLUSION

Throughout the creation of this thesis, a custom crucible dataset was developed using 260 crucible images taken from inside the VITRIOX® ELECTRIC machine and labelling them using the LabelImag [14] software. The selected three Convolutional Neural Network architectures were then configured from their downloaded pre-trained model and trained with the crucible dataset.

An industrial camera and lens necessary to fulfil the demands for machine learning models, focused on the requirements for the crucible position task, were then selected and mounted inside the VITRIOX® ELECTRIC machine to be used to test the different trained models.

A custom software called *FX_CruciblePositionChecker* was developed to test each trained model. Finally, an evaluation of the model's performance on the crucible position classification task were made. The accuracy and error rates were the main metrics used for this evaluation.

For the models trained to classify only the crucible handle position, it was observed that the results matched the initial assumptions made in section 4.2 The Base Networks. With the Faster R-CNN with Inception ResNet V2 model being the most accurate model by having an accuracy of 100% for all tested exposure times, the Faster R-CNN with ResNet-101 model being in the 'Sweet Spot' by demanding a lot less of computational power and still presenting considerable good results for all three exposure times, and the SSD with MobileNetV2 being the last accurate model with accuracy values below 60% for all three exposure times.

Now for the models trained to classify only the crucible axis position, the initial expectation for the performance of the models was not observed. The Faster R-CNN with Inception ResNet V2 showed accuracy values lower than the Faster R-CNN with ResNet-101, and in the tests made with a brighter light, the SSD model with MobileNetV2 showed a higher accuracy than the other two models.

For the crucible handle position, the classification is done as a standard object detection, meaning, the main task is to detect if the handle is above the platinum mould plate. For the crucible axis positions the task is a little more complicate, since the task is to differentiate the small changes of the axis direction on the crucible support. By default, CNN detection models are created with the goal of detecting an object regardless of its position on the picture, this could be the reason of the difference in the performance of the models for handle and axis position classification.

Besides that, the limitations encountered during the training of the Faster R-CNN with Inception ResNet V2 may have contributed to a lower accuracy of this model. Huang et al. [21] showed in their experiments that the reduced number of region proposals does not significantly harm the accuracy of the models, the same cannot be said about the number of training steps. For limitations of the computational power available, the Faster R-CNN with Inception ResNet V2 model was trained with less than half of the steps for the other two models, and this could be the reason for a lower accuracy of a more complex task as the axis position classification.

7.2 FUTURE WORK

The next step is the implementation of the chosen models into production. The deployment of a trained machine learning model is based on what the production environment will be, e.g., computer, remote server, mobile device, embedded Linux system, etc. TensorFlow has the advantage of supporting mobile, embedded, and IoT devices through *TensorFlow Lite*. That optimize the model for on-device use, making the running time less consuming as well as adding privacy, since no personal data leaves the device, and no internet connection is needed.

As discussed in chapter 3 The Dataset, the selection of the dataset used for the training of a convolutional neural network model is a very important step. For this thesis, a small crucible dataset was created to be used on a pre-trained model. For further research, with enough time to collect more and more crucible images, a big and diverse crucible dataset can be created.

Aside from the current crucible dataset, which already shows promising results, a customised model architecture focused on object orientation changes can be developed with a sufficiently large and diverse dataset. This will necessitate more training time, hardware power, and memory.

Due to a lack of processing resources, the resolution and Batch size were also reduced, as mentioned in section 4.4 Configuration of The Models. If more computing capacity is available, a custom model architecture with the required resolution, Bach size, and training steps can be trained from scratch. More research is needed to see how these parameters influence the output of CNN architectures for the Crucible dataset.

BIBLIOGRAPHY

1. Fluxana. *Product Catalogue XRF Application & Sample Preparation*. 10.11.2021]; Available from: https://fluxana.com/images/Catalogs/FLUXANA_Product_Catalog.pdf#page=13.
2. Allen, C.S., *Comparison of Depth image analysis methods for analytical machines controlled by a PLC*. 2021, Fluxana.
3. beh.digital. *beh.cam*. 10.11.2021]; Available from: https://beh.digital/en/beh-automated/.
4. Goodfellow, I., Y. Bengio, and A. Courville, *Deep Learning*. 2016: MIT Press.
5. Rosenblatt, F., *The perceptron: A probabilistic model for information storage and organization in the brain.* Psychological Review, 1958. **65**(6): p. 386-408.
6. Habibi Aghdam, H. and E. Jahani Heravi, *Guide to Convolutional Neural Networks: A Practical Application to Traffic-Sign Detection and Classification*. 2017, Cham: Springer International Publishing, Imprint: Springer.
7. Mou, L. and Z. Jin, *Tree-Based Convolutional Neural Networks: Principles and Applications*. SpringerBriefs in Computer Science. 2018, Singapore: Springer Singapore, Imprint: Springer.
8. Zhou, Z.-H., *Machine Learning*. 2021, Singapore: Springer Singapore, Imprint: Springer.
9. Skansi, S., *Introduction to Deep Learning: From Logical Calculus to Artificial Intelligence*. Undergraduate Topics in Computer Science. 2018, Cham: Springer International Publishing, Imprint: Springer.
10. Ioffe, S. and C. Szegedy, *Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift.* ArXiv, 2015. **abs/1502.03167**.
11. Srivastava, N., et al., *Dropout: a simple way to prevent neural networks from overfitting.* J. Mach. Learn. Res., 2014. **15**: p. 1929-1958.
12. Yosinski, J., et al., *How transferable are features in deep neural networks?* ArXiv, 2014. **abs/1411.1792**.
13. Lin, T.-Y., et al. *Microsoft coco: Common objects in context*. in *European conference on computer vision*. 2014. Springer.
14. Tzutalin. *LabelImg*. 2015 10.11.2021]; Git code]. Available from: https://github.com/tzutalin/labelImg.
15. TensorFlow. *TensorFlow 2 Detection Model Zoo*. 2021 16.11.2021]; Available from: https://github.com/tensorflow/models/blob/master/research/object_detection/g3doc/tf2_detection_zoo.md.
16. Abadi, M., et al., *Tensorflow: Large-scale machine learning on heterogeneous distributed systems.* arXiv preprint arXiv:1603.04467, 2016.
17. Van Rossum, G.a.D., Fred L., *Python 3 Reference Manual*. 2009: CreateSpace.
18. Bernico, M., *Deep Learning Quick Reference: Useful Hacks for Training and Optimizing Deep Neural Networks with TensorFlow and Keras*. 2018, Birmingham: Packt Publishing, Limited.

19. Kluyver, T., et al. *Jupyter Notebooks - a publishing format for reproducible computational workflows*. in *ELPUB*. 2016.

20. Millman, C.R.H.a.K.J., et al., *Array programming with NumPy.* Nature, 2020. **585**: p. 357--362.

21. Huang, J., et al., *Speed/Accuracy Trade-Offs for Modern Convolutional Object Detectors.* 2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR), 2017: p. 3296-3297.

22. Liu, W., et al. *SSD: Single Shot MultiBox Detector*. in *ECCV*. 2016.

23. Ren, S., et al., *Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks.* IEEE Transactions on Pattern Analysis and Machine Intelligence, 2015. **39**: p. 1137-1149.

24. Dai, J., et al. *R-fcn: Object detection via region-based fully convolutional networks*. in *Advances in neural information processing systems*. 2016.

25. Simonyan, K. and A. Zisserman, *Very deep convolutional networks for large-scale image recognition.* arXiv preprint arXiv:1409.1556, 2014.

26. Howard, A.G., et al., *Mobilenets: Efficient convolutional neural networks for mobile vision applications.* arXiv preprint arXiv:1704.04861, 2017.

27. He, K., et al., *Deep Residual Learning for Image Recognition.* 2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR), 2016: p. 770-778.

28. Szegedy, C., et al., *Rethinking the Inception Architecture for Computer Vision.* 2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR), 2016: p. 2818-2826.

29. Szegedy, C., et al. *Inception-v4, Inception-ResNet and the Impact of Residual Connections on Learning*. in *AAAI*. 2017.

30. Sandler, M., et al. *Mobilenetv2: Inverted residuals and linear bottlenecks*. in *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2018.

31. Girshick, R.B., et al., *Rich Feature Hierarchies for Accurate Object Detection and Semantic Segmentation.* 2014 IEEE Conference on Computer Vision and Pattern Recognition, 2014: p. 580-587.

32. Girshick, R. *Fast r-cnn*. in *Proceedings of the IEEE international conference on computer vision*. 2015.

33. Vladimirov, L. *TensorFlow 2 Object Detection API tutorial*. 2020 14.12.2021]; Revision 97dc1c92:[Available from: https://tensorflow-object-detection-api-tutorial.readthedocs.io/en/latest/install.html#tensorflow-object-detection-api-installation.

34. Yu, H., et al. *TensorFlow Model Garden*. 2020 14.12.2021]; Available from: https://github.com/tensorflow/models.

35. Chacon, S. and B. Straub, *Pro git*. 2nd Edition ed. 2014: Apress.

36. Google. *Protocol Buffers*. 2008 14.12.2021]; Google's data interchange format]. Available from: https://github.com/protocolbuffers/protobuf/releases.

37. Vladimirov, L. *Convert XML to record*. 2020 14.12.2021]; Available from: https://tensorflow-object-detection-api-tutorial.readthedocs.io/en/latest/training.html#convert-xml-to-record.

38. Vision, A. *Alvium 1800 U-500 DataSheet V1.7.2*. 16.11.2021]; Available from: https://cdn.alliedvision.com/fileadmin/pdf/de/Alvium_1800_U-500_DataSheet_V1.7.2_de.pdf.

39. optics, E. *4mm UC Series Fixed Focal Length Lens*. 16.11.2021]; Available from: https://www.edmundoptics.eu/p/4mm-uc-series-fixed-focal-length-lens/2966/.

40. Vision, A. *Vimba SDK*. 16.11.2021]; Available from: https://www.alliedvision.com/de/products/software/vimba-sdk/.

41. Lundh, F., *An introduction to tkinter*. 1999.

42. Saito, T. and M. Rehmsmeier, *The Precision-Recall Plot Is More Informative than the ROC Plot When Evaluating Binary Classifiers on Imbalanced Datasets*. PloS one, 2015. **10**: p. e0118432.

APPENDICES

APPENDIX A: CONFIGURATION FILE FOR SSD WITH MOBINETV2 MODELS

```
model {
  ssd {
    num_classes: 2
    image_resizer {
      fixed_shape_resizer {
        height: 320
        width: 320
      }
    }
    feature_extractor {
      type: "ssd_mobilenet_v2_fpn_keras"
      depth_multiplier: 1.0
      min_depth: 16
      conv_hyperparams {
        regularizer {
          l2_regularizer {
            weight: 4e-05
          }
        }
        initializer {
          random_normal_initializer {
            mean: 0.0
            stddev: 0.01
          }
        }
        activation: RELU_6
        batch_norm {
          decay: 0.997
          scale: true
          epsilon: 0.001
        }
      }
      use_depthwise: true
      override_base_feature_extractor_hyperparams: true
      fpn {
        min_level: 3
        max_level: 7
        additional_layer_depth: 128
      }
    }
    box_coder {
      faster_rcnn_box_coder {
        y_scale: 10.0
        x_scale: 10.0
```

```
    height_scale: 5.0
    width_scale: 5.0
  }
}
matcher {
  argmax_matcher {
    matched_threshold: 0.5
    unmatched_threshold: 0.5
    ignore_thresholds: false
    negatives_lower_than_unmatched: true
    force_match_for_each_row: true
    use_matmul_gather: true
  }
}
similarity_calculator {
  iou_similarity {
  }
}
box_predictor {
  weight_shared_convolutional_box_predictor {
    conv_hyperparams {
      regularizer {
        l2_regularizer {
          weight: 4e-05
        }
      }
      initializer {
        random_normal_initializer {
          mean: 0.0
          stddev: 0.01
        }
      }
      activation: RELU_6
      batch_norm {
        decay: 0.997
        scale: true
        epsilon: 0.001
      }
    }
    depth: 128
    num_layers_before_predictor: 4
    kernel_size: 3
    class_prediction_bias_init: -4.6
    share_prediction_tower: true
    use_depthwise: true
  }
}
anchor_generator {
  multiscale_anchor_generator {
    min_level: 3
```

```
        max_level: 7
        anchor_scale: 4.0
        aspect_ratios: 1.0
        aspect_ratios: 2.0
        aspect_ratios: 0.5
        scales_per_octave: 2
      }
    }
    post_processing {
      batch_non_max_suppression {
        score_threshold: 1e-08
        iou_threshold: 0.6
        max_detections_per_class: 100
        max_total_detections: 100
        use_static_shapes: false
      }
      score_converter: SIGMOID
    }
    normalize_loss_by_num_matches: true
    loss {
      localization_loss {
        weighted_smooth_l1 {
        }
      }
      classification_loss {
        weighted_sigmoid_focal {
          gamma: 2.0
          alpha: 0.25
        }
      }
      classification_weight: 1.0
      localization_weight: 1.0
    }
    encode_background_as_zeros: true
    normalize_loc_loss_by_codesize: true
    inplace_batchnorm_update: true
    freeze_batchnorm: false
  }
}
train_config {
  batch_size: 4
  data_augmentation_options {
    random_horizontal_flip {
    }
  }
  data_augmentation_options {
    random_crop_image {
      min_object_covered: 0.0
      min_aspect_ratio: 0.75
      max_aspect_ratio: 3.0
```

```
      min_area: 0.75
      max_area: 1.0
      overlap_thresh: 0.0
    }
  }
  sync_replicas: true
  optimizer {
    momentum_optimizer {
      learning_rate {
        cosine_decay_learning_rate {
          learning_rate_base: 0.08
          total_steps: 50000
          warmup_learning_rate: 0.026666
          warmup_steps: 1000
        }
      }
      momentum_optimizer_value: 0.9
    }
    use_moving_average: false
  }
  fine_tune_checkpoint: "Tensorflow/workspace/pre-trained-
models/ssd_mobilenet_v2_fpnlite_320x320_coco17_tpu-8/checkpoint/ckpt-0"
  num_steps: 50000
  startup_delay_steps: 0.0
  replicas_to_aggregate: 8
  max_number_of_boxes: 100
  unpad_groundtruth_tensors: false
  fine_tune_checkpoint_type: "detection"
  fine_tune_checkpoint_version: V2
}
train_input_reader {
  label_map_path: "Tensorflow/workspace/annotations/label_map.pbtxt"
  tf_record_input_reader {
    input_path: "Tensorflow/workspace/annotations/train.record"
  }
}
eval_config {
  metrics_set: "coco_detection_metrics"
  use_moving_averages: false
}
eval_input_reader {
  label_map_path: "Tensorflow/workspace/annotations/label_map.pbtxt"
  shuffle: false
  num_epochs: 1
  tf_record_input_reader {
    input_path: "Tensorflow/workspace/annotations/test.record"
  }
}
```

APPENDIX B: CONFIGURATION FILE FOR FASTER R-CNN WITH RESNET-101 MODELS

```
model {
  faster_rcnn {
    num_classes: 2
    image_resizer {
      keep_aspect_ratio_resizer {
        min_dimension: 320
        max_dimension: 320
        pad_to_max_dimension: true
      }
    }
    feature_extractor {
      type: 'faster_rcnn_resnet101_keras'
      batch_norm_trainable: true
    }
    first_stage_anchor_generator {
      grid_anchor_generator {
        scales: [0.25, 0.5, 1.0, 2.0]
        aspect_ratios: [0.5, 1.0, 2.0]
        height_stride: 16
        width_stride: 16
      }
    }
    first_stage_box_predictor_conv_hyperparams {
      op: CONV
      regularizer {
        l2_regularizer {
          weight: 0.0
        }
      }
      initializer {
        truncated_normal_initializer {
          stddev: 0.01
        }
      }
    }
    first_stage_nms_score_threshold: 0.0
    first_stage_nms_iou_threshold: 0.7
    first_stage_max_proposals: 100
    first_stage_localization_loss_weight: 2.0
    first_stage_objectness_loss_weight: 1.0
    initial_crop_size: 14
    maxpool_kernel_size: 2
    maxpool_stride: 2
    second_stage_box_predictor {
      mask_rcnn_box_predictor {
```

```
        use_dropout: false
        dropout_keep_probability: 1.0
        fc_hyperparams {
          op: FC
          regularizer {
            l2_regularizer {
              weight: 0.0
            }
          }
          initializer {
            variance_scaling_initializer {
              factor: 1.0
              uniform: true
              mode: FAN_AVG
            }
          }
        }
        share_box_across_classes: true
      }
    }
    second_stage_post_processing {
      batch_non_max_suppression {
        score_threshold: 0.0
        iou_threshold: 0.6
        max_detections_per_class: 100
        max_total_detections: 300
      }
      score_converter: SOFTMAX
    }
    second_stage_localization_loss_weight: 2.0
    second_stage_classification_loss_weight: 1.0
    use_static_shapes: true
    use_matmul_crop_and_resize: true
    clip_anchors_to_image: true
    use_static_balanced_label_sampler: true
    use_matmul_gather_in_matcher: true
  }
}

train_config: {
  batch_size: 4
  sync_replicas: true
  startup_delay_steps: 0
  replicas_to_aggregate: 8
  num_steps: 25000
  optimizer {
    momentum_optimizer: {
      learning_rate: {
        cosine_decay_learning_rate {
          learning_rate_base: .04
```

```
          total_steps: 25000
          warmup_learning_rate: .013333
          warmup_steps: 2000
        }
      }
      momentum_optimizer_value: 0.9
    }
    use_moving_average: false
  }
  fine_tune_checkpoint_version: V2
  fine_tune_checkpoint: "Tensorflow/workspace/pre-trained-
models/faster_rcnn_resnet101_v1_640x640_coco17_tpu-8/checkpoint/ckpt-0"
  fine_tune_checkpoint_type: "detection"
  data_augmentation_options {
    random_horizontal_flip {
    }
  }

  max_number_of_boxes: 100
  unpad_groundtruth_tensors: false
  use_bfloat16: false  # works only on TPUs
}

train_input_reader: {
  label_map_path: "Tensorflow/workspace/annotations/label_map.pbtxt"
  tf_record_input_reader {
    input_path: "Tensorflow/workspace/annotations/train.record"
  }
}

eval_config: {
  metrics_set: "coco_detection_metrics"
  use_moving_averages: false
  batch_size: 1;
}
eval_input_reader: {
  label_map_path: "Tensorflow/workspace/annotations/label_map.pbtxt"
  shuffle: false
  num_epochs: 1
  tf_record_input_reader {
    input_path: "Tensorflow/workspace/annotations/test.record"
  }
}
```

APPENDIX C: CONFIGURATION FILE FOR FASTER R-CNN WITH INCEPTION RESNET V2 MODELS

```
# Faster R-CNN with Inception Resnet v2 (no atrous)
# Sync-trained on COCO with batch size 64 (800x1333 resolution)
# Initialized from Imagenet classification checkpoint
# TF2-Compatible
# Train on TPU (w/2x2 topology)
#
# Achieves 39.85 mAP on COCO17

model {
  faster_rcnn {
    num_classes: 2
    image_resizer {
      fixed_shape_resizer {
        height: 320
        width: 320
      }
    }
    feature_extractor {
      type: 'faster_rcnn_inception_resnet_v2_keras'
      batch_norm_trainable: true
    }
    first_stage_anchor_generator {
      grid_anchor_generator {
        scales: [0.25, 0.5, 1.0, 2.0]
        aspect_ratios: [0.5, 1.0, 2.0]
        height_stride: 16
        width_stride: 16
      }
    }
    first_stage_box_predictor_conv_hyperparams {
      op: CONV
      regularizer {
        l2_regularizer {
          weight: 0.0
        }
      }
      initializer {
        truncated_normal_initializer {
          stddev: 0.01
        }
      }
    }
    first_stage_nms_score_threshold: 0.0
    first_stage_nms_iou_threshold: 0.7
    first_stage_max_proposals: 100
    first_stage_localization_loss_weight: 2.0
    first_stage_objectness_loss_weight: 1.0
```

```
    initial_crop_size: 17
    maxpool_kernel_size: 1
    maxpool_stride: 1
    second_stage_box_predictor {
      mask_rcnn_box_predictor {
        use_dropout: false
        dropout_keep_probability: 1.0
        fc_hyperparams {
          op: FC
          regularizer {
            l2_regularizer {
              weight: 0.0
            }
          }
          initializer {
            variance_scaling_initializer {
              factor: 1.0
              uniform: true
              mode: FAN_AVG
            }
          }
        }
        share_box_across_classes: true
      }
    }
    second_stage_post_processing {
      batch_non_max_suppression {
        score_threshold: 0.0
        iou_threshold: 0.6
        max_detections_per_class: 100
        max_total_detections: 100
      }
      score_converter: SOFTMAX
    }
    second_stage_localization_loss_weight: 2.0
    second_stage_classification_loss_weight: 1.0
    use_static_shapes: true
    use_matmul_crop_and_resize: true
    clip_anchors_to_image: true
    use_static_balanced_label_sampler: true
    use_matmul_gather_in_matcher: true
  }
}

train_config: {
  batch_size: 4
  sync_replicas: true
  startup_delay_steps: 0
  replicas_to_aggregate: 8
  num_steps: 100000
```

```
  optimizer {
    momentum_optimizer: {
      learning_rate: {
        cosine_decay_learning_rate {
          learning_rate_base: .16
          total_steps: 100000
          warmup_learning_rate: 0
          warmup_steps: 2500
        }
      }
      momentum_optimizer_value: 0.9
    }
    use_moving_average: false
  }
  fine_tune_checkpoint_version: V2
  fine_tune_checkpoint: "Tensorflow/workspace/pre-trained-
models/faster_rcnn_inception_resnet_v2_640x640_coco17_tpu-8/checkpoint/ckpt-0"
  fine_tune_checkpoint_type: "detection"
  data_augmentation_options {
    random_horizontal_flip {
    }
  }
  data_augmentation_options {
    random_adjust_hue {
    }
  }

  data_augmentation_options {
    random_adjust_contrast {
    }
  }

  data_augmentation_options {
    random_adjust_saturation {
    }
  }

  data_augmentation_options {
     random_square_crop_by_scale {
      scale_min: 0.6
      scale_max: 1.3
    }
  }
  max_number_of_boxes: 100
  unpad_groundtruth_tensors: false
  use_bfloat16: false  # works only on TPUs
}
train_input_reader: {
  label_map_path: "Tensorflow/workspace/annotations/label_map.pbtxt"
  tf_record_input_reader {
```

```
    input_path: "Tensorflow/workspace/annotations/train.record"
  }
}

eval_config: {
  metrics_set: "coco_detection_metrics"
  use_moving_averages: false
  batch_size: 1;
}

eval_input_reader: {
  label_map_path: "Tensorflow/workspace/annotations/label_map.pbtxt"
  shuffle: false
  num_epochs: 1
  tf_record_input_reader {
    input_path: "Tensorflow/workspace/annotations/test.record"
  }
}
```

APPENDIX D: FX_CRUCIBLEPOSITIONCHECKER.PY FULL CODE

```
ANNOTATION_PATH = 'annotations' #Name of the folder where the label and record
files are saved
MODEL_PATH = 'models' #Name of the folder where the models are saved
CONFIG_PATH_AXIS = MODEL_PATH+'/my_faster_rcnn_resnet101_v1/pipeline.config'
#path of the configuration file of the axis model
CONFIG_PATH_HANDLE =
MODEL_PATH+'/my_faster_rcnn_inception_resnet_v2/pipeline.config' #path of the
configuration file of the handle model
CHECKPOINT_PATH_AXIS = MODEL_PATH+'/my_faster_rcnn_resnet101_v1/' #Name of the
folder where the axis model's checkpoints are saved
CHECKPOINT_PATH_HANDLE = MODEL_PATH+'/my_faster_rcnn_inception_resnet_v2/'
#Name of the folder where the handle model's checkpoints are saved

import tensorflow as tf #imports tensorflow library

#Imports Google's object detection API
from object_detection.utils import config_util
from object_detection.utils import label_map_util
from object_detection.utils import visualization_utils as viz_utils
from object_detection.builders import model_builder

import cv2 #Imports OpenCV for image data handling
#Import libraries that handle system files and variables
import os
import sys

from typing import Optional #Import typing libraries
import numpy as np #Imports library for handle array manipulation
#Imports the industrial camera library
from vimba import*
from vimba.c_binding.vimba_common import VmbPixelFormat
#Import Tkinter libraries for GUI creation
import tkinter as tk
from tkinter import font as tkFont
from PIL import ImageTk, Image


#Get which position classification task need to be performed (1=axis;2=handle)
classification_type = int(input("Enter which position classification task need
to be performed (1=axis;2=handle): "))

# Load pipeline config and build a detection model
if classification_type == 1:
    configs = config_util.get_configs_from_pipeline_file(CONFIG_PATH_AXIS)
elif classification_type == 2:
    configs = config_util.get_configs_from_pipeline_file(CONFIG_PATH_HANDLE)
```

```
detection_model = model_builder.build(model_config=configs['model'],
is_training=False)

# Restore checkpoint
ckpt = tf.compat.v2.train.Checkpoint(model=detection_model)
if classification_type == 1:
    ckpt.restore(os.path.join(CHECKPOINT_PATH_AXIS, 'ckpt-
11')).expect_partial()
elif classification_type == 2:
    ckpt.restore(os.path.join(CHECKPOINT_PATH_HANDLE, 'ckpt-
5')).expect_partial()

#Function where the models makes the epredictions
@tf.function
def detect_fn(image):
    image, shapes = detection_model.preprocess(image)
    prediction_dict = detection_model.predict(image, shapes)
    detections = detection_model.postprocess(prediction_dict, shapes)
    return detections

#pass the classes and their respective Id
category_index =
label_map_util.create_category_index_from_labelmap(ANNOTATION_PATH+'/label_map
.pbtxt')

#Get the current camera frame
def frame_handler(cam , frame ):
    cam.queue_frame(frame)


#aborts the camera capturing
def abort(reason: str, return_code: int = 1, usage: bool = False):
    sys.exit(return_code)

#Function to get the camera id from command line arguments
def parse_args() -> Optional[35]:
    args = sys.argv[1:]
    argc = len(args)

    for arg in args:
        if arg in ('/h', '-h'):
            sys.exit(0)

    if argc > 1:
        abort(reason="Invalid number of arguments. Abort.", return_code=2,
usage=True)

    return None if argc == 0 else args[0]
```

```
#function to get the camera id from the system by the id, if id was not
passed, it gets the first camera found in the system
def get_camera(camera_id: Optional[35]) -> Camera:
    with Vimba.get_instance() as vimba:
        if camera_id:
            try:
                return vimba.get_camera_by_id(camera_id)
            except VimbaCameraError:
                abort('Failed to access Camera \'{}\'.
Abort.'.format(camera_id))
        else:
            cams = vimba.get_all_cameras()
            if not cams:
                abort('No Cameras accessible. Abort.')

            return cams[0]

#saves the camera id into a variable
cam_id = parse_args()


display_option = "1" #Set the display option of the frame (1=current camera
frame;2=predicted result with drawn bounding boxes)
predict = False #Flag to indicate if the software should make a prediction of
the current frame of not
ImagesPath = "images" #folder where the images for the buttons are saved

#main function of the program
if __name__ == "__main__":

    #get the vimba camera instance
    with Vimba.get_instance():
        with get_camera(cam_id) as cam:

            # Load camera settings from file.
            settings_file = 'settings.xml'
            cam.load_settings(settings_file, PersistType.All)

    #create th emain window of the appliation
    root = tk.Tk()
    root.title("FX_CruciblePositionChecker")


    root.geometry('{}x{}'.format(1000, 800)) #set the application window size
    #create the main frame of the application
    main_frame = tk.Frame(root)
    main_frame.pack(fill=tk.BOTH)
    #cretae the label where the camera frames will be displayed
    lmain = tk.Label(main_frame)
    lmain.grid(row=0,column=0)
```

```
    #create the inner frame where the buttons will be inserted
    inner_frame = tk.Frame(main_frame)
    inner_frame.grid(row=0,column=1)

    #define the text font for the buttons
    mainFont = tkFont.Font(family="Arial", size=20, weight="bold")

    #predict button function to set the application mode to prediction
    def PredictBtnClick():
       global display_option,predict
       display_option = "2"
       predict = True

    #Create the predict button and assign to its function
    PredictBtn =
tk.Button(inner_frame,text="Check",command=PredictBtnClick,font=mainFont)
    PredictBtn.grid(row=0,column=0,padx=(20,0))

    #Reset button function to set the application mode to normal view
    def ResetBtnClick():
       global display_option
       display_option = "1"

    #Create the reset button and assign to its function
    ResetBtn
=  tk.Button(inner_frame,text="Reset",command=ResetBtnClick,font=mainFont)
    ResetBtn.grid(row=1,column=0,padx=(20,0))


    #Get image of "empy button" to be displayed when there is no prediction
displayed
    StatussBtnImagepath = ImagesPath + "\empty button.png"
    StatussBtnImage =
ImageTk.PhotoImage(Image.open(StatussBtnImagepath).resize((140, 140),
Image.ANTIALIAS))
    #Create the button that will display the status of the picture
frame('empty'= no prediction are been displayed; ‘green'= the model predicted
as 'Good’; ‘red'= the model predicted as 'Bad')
    StatussBtn = tk.Button(inner_frame,width=140,height=140)
    StatussBtn.grid(row=0,column=1,padx=(20,0),rowspan=2)

    #Get image of "Green button" to be displayed when the model is showing a
good position predction
    GreenBtnImagepath = ImagesPath + "\green button.png"
    GreenBtnImage =
ImageTk.PhotoImage(Image.open(GreenBtnImagepath).resize((140, 140),
Image.ANTIALIAS))
    #Get image of "Red button" to be displayed when the model is showing a bad
position predction
    RedBtnImagepath = ImagesPath + "\\red button.png"
```

```
    RedBtnImage = ImageTk.PhotoImage(Image.open(RedBtnImagepath).resize((140,
140), Image.ANTIALIAS))

    #function where the frames are displayed
    def show_frame():
        with Vimba.get_instance():
            with get_camera(cam_id) as cam:

                # Aquire single frame synchronously
                frame = cam.get_frame ()
                image_size = frame.get_image_size()
                if image_size != 0:
                    frame.convert_pixel_format(PixelFormat.Rgb8)
                    global predict
                    #If the display flag is set to 1, the application shows
the current frame of the camera without any predictions
                    if display_option == "1":
                        try:
                            image_np = frame.as_numpy_ndarray()
                            cv2image = cv2.resize(image_np, (648, 486))
                            img = Image.fromarray(cv2image)
                            imgtk = ImageTk.PhotoImage(image=img)
                            lmain.imgtk = imgtk
                            lmain.configure(image=imgtk)
                            StatussBtn.config(image=StatussBtnImage)
                        except Exception as err:
                            print(err)
                    #If the display flag is set to 2, the application makes a
prediction and shows the results in the image
                    elif display_option == "2" and predict == True:
                        try:
                            image_np = frame.as_numpy_ndarray()
                            input_tensor =
tf.convert_to_tensor(np.expand_dims(image_np, 0), dtype=tf.float32)
                            detections = detect_fn(input_tensor)

                            num_detections =
int(detections.pop('num_detections'))
                            detections = {key: value[0,
:num_detections].numpy()
                                          for key, value in detections.items()}
                            detections['num_detections'] = num_detections

                            # detection_classes should be ints.
                            detections['detection_classes'] =
detections['detection_classes'].astype(np.int64)
                            label_id_offset = 1
                            image_np_with_detections = image_np.copy()
                            image_np_with_detections =
cv2.resize(image_np_with_detections, (648, 486))
```

```
                            #function to drawn the predicted boxes in the
current image
                            viz_utils.visualize_boxes_and_labels_on_image_arra
y(
                                image_np_with_detections,
                                detections['detection_boxes'],
                                detections['detection_classes']+label_id_offse
t,
                                detections['detection_scores'],
                                category_index,
                                use_normalized_coordinates=True,
                                max_boxes_to_draw=5,
                                min_score_thresh=.3,
                                agnostic_mode=False)

                            cv2image =image_np_with_detections
                            img = Image.fromarray(cv2image)
                            imgtk = ImageTk.PhotoImage(image=img)
                            lmain.imgtk = imgtk
                            lmain.configure(image=imgtk)
                            predict = False

                            #If both axes where detected as a good position,
the status button is set to green, anything else, the status button is set to
red
                            global RedBtnImage,GreenBtnImage
                            if
category_index[detections['detection_classes'][0]+label_id_offset]['name'] ==
"Good" and
category_index[detections['detection_classes'][1]+label_id_offset]['name'] ==
"Good" and classification_type == 1:
                                StatussBtn.config(image=GreenBtnImage)
                            elif
category_index[detections['detection_classes'][0]+label_id_offset]['name'] ==
"Good" and classification_type == 2:
                                StatussBtn.config(image=GreenBtnImage)
                            else:
                                StatussBtn.config(image=RedBtnImage)


                        except Exception as err:
                            print(err)

        lmain.after(10, show_frame)



show_frame()
root.mainloop()
```

APPENDIX E: RESULTS FOR SSD WITH MOBILENETV2 ONLY AXIS MODEL, $exposure = 10002{,}435\ \mu s$

True Class	Predict Class	Upper axis score	Lower axis score	Image
Good	Bad	0	94,00%	good1.png
Good	Good	0	94,00%	good2.png
Good	Bad	0	53,00%	good3.png
Good	Bad	0	94,00%	good4.png
Good	Bad	0	95,00%	good5.png
Good	Bad	0	93,00%	good6.png
Good	Bad	0	73,00%	good7.png
Good	Bad	0	94,00%	good8.png
Good	Bad	0	94,00%	good9.png
Good	Bad	0	75,00%	good10.png
Good	Bad	0	95,00%	good11.png
Good	Bad	0	95,00%	good12.png
Good	Bad	0	95,00%	good13.png
Good	Bad	0	45,00%	good14.png
Good	Bad	0	93,00%	good15.png
Good	Bad	0	95,00%	good16.png
Good	Bad	0	95,00%	good17.png
Good	Bad	0	94,00%	good18.png
Good	Bad	0	95,00%	good19.png
Good	Bad	0	95,00%	good20.png
Good	Bad	0	92,00%	good21.png
Good	Bad	0	42,00%	good22.png
Good	Bad	0	38,00%	good23.png
Good	Bad	0	92,00%	good24.png
Good	Bad	0	95,00%	good25.png
Good	Good	0	95,00%	good26.png
Good	Bad	0	94,00%	good27.png
Good	Bad	0	94,00%	good28.png
Good	Bad	0	95,00%	good29.png
Good	Bad	0	30,00%	good30.png

Good	Bad	0	89,00%	good31.png
Good	Bad	0	19,00%	good32.png
Good	Bad	0	95,00%	good33.png
Good	Bad	0	29,00%	good34.png
Good	Bad	15,00%	95,00%	good35.png
Good	Bad	0	86,00%	good36.png
Good	Bad	0	94,00%	good37.png
Good	Bad	12,00%	92,00%	good38.png
Good	Bad	0	67,00%	good39.png
Good	Bad	0	81,00%	good40.png
Good	Bad	0	95,00%	good41.png
Good	Bad	0	30,00%	good42.png
Good	Bad	13,00%	85,00%	good43.png
Good	Bad	0	92,00%	good44.png
Good	Bad	0	93,00%	good45.png
Good	Bad	0	13,00%	good46.png
Good	Good	0	95,00%	good47.png
Good	Bad	0	94,00%	good48.png
Good	Bad	0	95,00%	good49.png
Good	Bad	0	94,00%	good50.png
Bad	Bad	0	90,00%	bad1.png
Bad	Bad	0	81,00%	bad2.png
Bad	Bad	0	88,00%	bad3.png
Bad	Bad	0	85,00%	bad4.png
Bad	Bad	0	88,00%	bad5.png
Bad	Bad	0	85,00%	bad6.png
Bad	Bad	0	81,00%	bad7.png
Bad	Bad	0	87,00%	bad8.png
Bad	Bad	0	76,00%	bad9.png
Bad	Bad	0	23,00%	bad10.png
Bad	Bad	0	29,00%	bad11.png
Bad	Bad	0	53,00%	bad12.png
Bad	Bad	0	77,00%	bad13.png
Bad	Bad	0	77,00%	bad14.png
Bad	Bad	0	81,00%	bad15.png

Bad	Bad	0	85,00%	bad16.png
Bad	Bad	0	94,00%	bad17.png
Bad	Bad	0	94,00%	bad18.png
Bad	Bad	0	94,00%	bad19.png
Bad	Bad	0	90,00%	bad20.png
Bad	Bad	0	90,00%	bad21.png
Bad	Bad	0	92,00%	bad22.png
Bad	Bad	0	89,00%	bad23.png
Bad	Bad	0	90,00%	bad24.png
Bad	Bad	0	88,00%	bad25.png
Bad	Bad	0	84,00%	bad26.png
Bad	Bad	0	84,00%	bad27.png
Bad	Bad	0	88,00%	bad28.png
Bad	Bad	0	91,00%	bad29.png
Bad	Bad	0	87,00%	bad30.png
Bad	Bad	0	86,00%	bad31.png
Bad	Bad	0	80,00%	bad32.png
Bad	Bad	0	92,00%	bad33.png
Bad	Bad	0	77,00%	bad34.png
Bad	Bad	0	86,00%	bad35.png
Bad	Bad	0	89,00%	bad36.png
Bad	Bad	0	82,00%	bad37.png
Bad	Bad	0	92,00%	bad38.png
Bad	Bad	0	84,00%	bad39.png
Bad	Bad	0	89,00%	bad40.png
Bad	Bad	0	90,00%	bad41.png
Bad	Bad	0	92,00%	bad42.png
Bad	Bad	0	94,00%	bad43.png
Bad	Bad	0	93,00%	bad44.png
Bad	Bad	0	81,00%	bad45.png
Bad	Bad	0	53,00%	bad46.png
Bad	Bad	0	73,00%	bad47.png
Bad	Bad	0	44,00%	bad48.png
Bad	Bad	0	92,00%	bad49.png
Bad	Bad	0	76,00%	bad50.png

APPENDIX F: RESULTS FOR SSD WITH MOBILENETV2 ONLY AXIS MODEL, $exposure = 21857{,}652\ \mu s$

True Class	Predict Class	Upper axis score	Lower axis score	Image
Good	Good	71,00%	96,00%	good1.png
Good	Good	65,00%	92,00%	good2.png
Good	Good	60,00%	97,00%	good3.png
Good	Good	23,00%	95,00%	good4.png
Good	Good	82,00%	95,00%	good5.png
Good	Good	72,00%	96,00%	good6.png
Good	Good	94,00%	89,00%	good7.png
Good	Good	73,00%	95,00%	good8.png
Good	Good	82,00%	95,00%	good9.png
Good	Good	73,00%	96,00%	good10.png
Good	Good	92,00%	87,00%	good11.png
Good	Good	84,00%	95,00%	good12.png
Good	Good	87,00%	95,00%	good13.png
Good	Good	90,00%	86,00%	good14.png
Good	Good	67,00%	95,00%	good15.png
Good	Good	82,00%	95,00%	good16.png
Good	Bad	12,00%	93,00%	good17.png
Good	Good	38,00%	96,00%	good18.png
Good	Good	75,00%	74,00%	good19.png
Good	Bad	42,00%	57,00%	good20.png
Good	Good	65,00%	93,00%	good21.png
Good	Good	58,00%	57,00%	good22.png
Good	Good	84,00%	80,00%	good23.png
Good	Good	64,00%	97,00%	good24.png
Good	Good	68,00%	95,00%	good25.png
Good	Good	43,00%	94,00%	good26.png
Good	Good	79,00%	94,00%	good27.png
Good	Good	63,00%	86,00%	good28.png
Good	Good	0	96,00%	good29.png
Good	Good	58,00%	96,00%	good30.png

Good	Bad	30,00%	41,00%	good31.png
Good	Good	46,00%	97,00%	good32.png
Good	Good	43,00%	90,00%	good33.png
Good	Good	83,00%	96,00%	good34.png
Good	Bad	0	56,00%	good35.png
Good	Good	61,00%	92,00%	good36.png
Good	Good	81,00%	96,00%	good37.png
Good	Good	62,00%	0	good38.png
Good	Good	85,00%	96,00%	good39.png
Good	Good	78,00%	97,00%	good40.png
Good	Good	81,00%	96,00%	good41.png
Good	Bad	73,00%	41,00%	good42.png
Good	Good	73,00%	96,00%	good43.png
Good	Good	83,00%	96,00%	good44.png
Good	Good	23,00%	89,00%	good45.png
Good	Bad	56,00%	45,00%	good46.png
Good	Good	62,00%	89,00%	good47.png
Good	Good	87,00%	96,00%	good48.png
Good	Good	48,00%	94,00%	good49.png
Good	Bad	24,00%	49,00%	good50.png
Bad	Bad	45,00%	93,00%	bad1.png
Bad	Bad	78,00%	94,00%	bad2.png
Bad	Bad	55,00%	85,00%	bad3.png
Bad	Bad	74,00%	94,00%	bad4.png
Bad	Bad	44,00%	92,00%	bad5.png
Bad	Bad	81,00%	91,00%	bad6.png
Bad	Bad	89,00%	85,00%	bad7.png
Bad	Bad	71,00%	91,00%	bad8.png
Bad	Bad	73,00%	65,00%	bad9.png
Bad	Bad	76,00%	77,00%	bad10.png
Bad	Bad	80,00%	70,00%	bad11.png
Bad	Bad	58,00%	66,00%	bad12.png
Bad	Bad	26,00%	95,00%	bad13.png
Bad	Bad	63,00%	93,00%	bad14.png
Bad	Bad	67,00%	95,00%	bad15.png

Bad	Bad	63,00%	91,00%	bad16.png
Bad	Bad	44,00%	93,00%	bad17.png
Bad	Bad	64,00%	94,00%	bad18.png
Bad	Bad	73,00%	93,00%	bad19.png
Bad	Bad	40,00%	93,00%	bad20.png
Bad	Bad	68,00%	92,00%	bad21.png
Bad	Bad	63,00%	90,00%	bad22.png
Bad	Bad	73,00%	94,00%	bad23.png
Bad	Bad	51,00%	93,00%	bad24.png
Bad	Bad	40,00%	94,00%	bad25.png
Bad	Bad	63,00%	95,00%	bad26.png
Bad	Bad	53,00%	89,00%	bad27.png
Bad	Bad	45,00%	93,00%	bad28.png
Bad	Bad	58,00%	94,00%	bad29.png
Bad	Bad	0	89,00%	bad30.png
Bad	Bad	68,00%	85,00%	bad31.png
Bad	Bad	22,00%	93,00%	bad32.png
Bad	Bad	0	91,00%	bad33.png
Bad	Bad	41,00%	93,00%	bad34.png
Bad	Bad	64,00%	87,00%	bad35.png
Bad	Bad	28,00%	91,00%	bad36.png
Bad	Bad	0	91,00%	bad37.png
Bad	Bad	72,00%	93,00%	bad38.png
Bad	Bad	30,00%	87,00%	bad39.png
Bad	Bad	0	90,00%	bad40.png
Bad	Bad	37,00%	90,00%	bad41.png
Bad	Bad	0	89,00%	bad42.png
Bad	Bad	0	80,00%	bad43.png
Bad	Bad	63,00%	81,00%	bad44.png
Bad	Bad	30,00%	85,00%	bad45.png
Bad	Bad	60,00%	93,00%	bad46.png
Bad	Bad	79,00%	82,00%	bad47.png
Bad	Bad	41,00%	87,00%	bad48.png
Bad	Bad	43,00%	87,00%	bad49.png
Bad	Bad	33,00%	90,00%	bad50.png

APPENDIX G: RESULTS FOR SSD WITH MOBILENETV2 ONLY AXIS MODEL, $exposure = 32002{,}609\ \mu s$

True Class	Predict Class	Upper axis score	Lower axis score	Image
Good	Good	96,00%	82,00%	good1.png
Good	Good	84,00%	20,00%	good2.png
Good	Good	90,00%	96,00%	good3.png
Good	Good	95,00%	91,00%	good4.png
Good	Good	94,00%	77,00%	good5.png
Good	Good	84,00%	95,00%	good6.png
Good	Good	81,00%	17,00%	good7.png
Good	Good	95,00%	86,00%	good8.png
Good	Good	94,00%	83,00%	good9.png
Good	Good	94,00%	84,00%	good10.png
Good	Good	89,00%	95,00%	good11.png
Good	Good	94,00%	87,00%	good12.png
Good	Good	94,00%	89,00%	good13.png
Good	Good	93,00%	19,00%	good14.png
Good	Good	84,00%	96,00%	good15.png
Good	Good	96,00%	89,00%	good16.png
Good	Good	94,00%	90,00%	good17.png
Good	Good	94,00%	74,00%	good18.png
Good	Good	92,00%	90,00%	good19.png
Good	Good	95,00%	85,00%	good20.png
Good	Good	96,00%	86,00%	good21.png
Good	Good	93,00%	19,00%	good22.png
Good	Good	91,00%	96,00%	good23.png
Good	Good	96,00%	87,00%	good24.png
Good	Good	92,00%	87,00%	good25.png
Good	Good	94,00%	94,00%	good26.png
Good	Good	96,00%	76,00%	good27.png
Good	Good	95,00%	90,00%	good28.png
Good	Good	95,00%	90,00%	good29.png
Good	Good	94,00%	17,00%	good30.png

Good	Good	90,00%	96,00%	good31.png
Good	Good	96,00%	92,00%	good32.png
Good	Good	96,00%	82,00%	good33.png
Good	Good	96,00%	72,00%	good34.png
Good	Good	96,00%	86,00%	good35.png
Good	Good	90,00%	92,00%	good36.png
Good	Good	94,00%	87,00%	good37.png
Good	Good	94,00%	16,00%	good38.png
Good	Good	86,00%	96,00%	good39.png
Good	Good	96,00%	90,00%	good40.png
Good	Good	89,00%	86,00%	good41.png
Good	Good	95,00%	32,00%	good42.png
Good	Good	93,00%	92,00%	good43.png
Good	Good	94,00%	87,00%	good44.png
Good	Good	94,00%	87,00%	good45.png
Good	Good	91,00%	0	good46.png
Good	Good	94,00%	89,00%	good47.png
Good	Good	81,00%	96,00%	good48.png
Good	Good	91,00%	92,00%	good49.png
Good	Good	94,00%	89,00%	good50.png
Bad	Bad	59,00%	25,00%	bad1.png
Bad	Bad	96,00%	70,00%	bad2.png
Bad	Bad	76,00%	77,00%	bad3.png
Bad	Bad	86,00%	42,00%	bad4.png
Bad	Bad	71,00%	34,00%	bad5.png
Bad	Bad	93,00%	50,00%	bad6.png
Bad	Bad	84,00%	80,00%	bad7.png
Bad	Bad	93,00%	29,00%	bad8.png
Bad	Bad	89,00%	16,00%	bad9.png
Bad	Bad	96,00%	0	bad10.png
Bad	Bad	66,00%	12,00%	bad11.png
Bad	Bad	96,00%	11,00%	bad12.png
Bad	Bad	68,00%	82,00%	bad13.png
Bad	Bad	90,00%	90,00%	bad14.png
Bad	Bad	91,00%	89,00%	bad15.png

Bad	Bad	82,00%	91,00%	bad16.png
Bad	Bad	78,00%	46,00%	bad17.png
Bad	Bad	95,00%	85,00%	bad18.png
Bad	Bad	78,00%	82,00%	bad19.png
Bad	Bad	92,00%	36,00%	bad20.png
Bad	Bad	56,00%	30,00%	bad21.png
Bad	Bad	95,00%	62,00%	bad22.png
Bad	Bad	84,00%	58,00%	bad23.png
Bad	Bad	92,00%	30,00%	bad24.png
Bad	Bad	75,00%	83,00%	bad25.png
Bad	Bad	91,00%	94,00%	bad26.png
Bad	Bad	96,00%	84,00%	bad27.png
Bad	Bad	93,00%	90,00%	bad28.png
Bad	Bad	87,00%	19,00%	bad29.png
Bad	Bad	96,00%	0	bad30.png
Bad	Bad	86,00%	14,00%	bad31.png
Bad	Bad	95,00%	15,00%	bad32.png
Bad	Bad	80,00%	64,00%	bad33.png
Bad	Bad	97,00%	81,00%	bad34.png
Bad	Bad	90,00%	54,00%	bad35.png
Bad	Bad	93,00%	33,00%	bad36.png
Bad	Bad	68,00%	46,00%	bad37.png
Bad	Bad	94,00%	76,00%	bad38.png
Bad	Bad	86,00%	35,00%	bad39.png
Bad	Bad	90,00%	59,00%	bad40.png
Bad	Bad	89,00%	22,00%	bad41.png
Bad	Bad	96,00%	12,00%	bad42.png
Bad	Bad	59,00%	17,00%	bad43.png
Bad	Bad	92,00%	24,00%	bad44.png
Bad	Bad	64,00%	92,00%	bad45.png
Bad	Bad	90,00%	92,00%	bad46.png
Bad	Bad	86,00%	89,00%	bad47.png
Bad	Bad	74,00%	94,00%	bad48.png
Bad	Bad	74,00%	59,00%	bad49.png
Bad	Bad	96,00%	55,00%	bad50.png

APPENDIX H: RESULTS FOR FASTER R-CNN WITH RESNET-101 ONLY AXIS MODEL, $exposure = 10002{,}435\,\mu s$

True Class	Predict Class	Upper axis score	Lower axis score	Image
Good	Good	100,00%	100,00%	good1.png
Good	Good	88,00%	100,00%	good2.png
Good	Good	95,00%	95,00%	good3.png
Good	Good	100,00%	100,00%	good4.png
Good	Good	100,00%	100,00%	good5.png
Good	Good	69,00%	100,00%	good6.png
Good	Good	97,00%	99,00%	good7.png
Good	Good	100,00%	100,00%	good8.png
Good	Good	100,00%	100,00%	good9.png
Good	Good	98,00%	100,00%	good10.png
Good	Good	99,00%	100,00%	good11.png
Good	Good	99,00%	100,00%	good12.png
Good	Good	99,00%	100,00%	good13.png
Good	Good	91,00%	98,00%	good14.png
Good	Good	36,00%	100,00%	good15.png
Good	Good	100,00%	100,00%	good16.png
Good	Good	99,00%	100,00%	good17.png
Good	Good	92,00%	100,00%	good18.png
Good	Good	98,00%	100,00%	good19.png
Good	Good	97,00%	100,00%	good20.png
Good	Good	100,00%	99,00%	good21.png
Good	Good	96,00%	100,00%	good22.png
Good	Good	89,00%	99,00%	good23.png
Good	Good	97,00%	100,00%	good24.png
Good	Good	100,00%	100,00%	good25.png
Good	Good	72,00%	100,00%	good26.png
Good	Good	100,00%	100,00%	good27.png
Good	Good	100,00%	100,00%	good28.png
Good	Good	100,00%	99,00%	good29.png
Good	Good	85,00%	95,00%	good30.png

Good	Good	91,00%	100,00%	good31.png
Good	Good	99,00%	100,00%	good32.png
Good	Good	100,00%	100,00%	good33.png
Good	Good	22,00%	97,00%	good34.png
Good	Good	70,00%	100,00%	good35.png
Good	Good	99,00%	100,00%	good36.png
Good	Good	100,00%	100,00%	good37.png
Good	Good	69,00%	100,00%	good38.png
Good	Good	95,00%	99,00%	good39.png
Good	Good	100,00%	100,00%	good40.png
Good	Good	97,00%	100,00%	good41.png
Good	Good	81,00%	98,00%	good42.png
Good	Good	55,00%	100,00%	good43.png
Good	Good	99,00%	100,00%	good44.png
Good	Good	99,00%	100,00%	good45.png
Good	Good	91,00%	97,00%	good46.png
Good	Good	57,00%	100,00%	good47.png
Good	Good	99,00%	100,00%	good48.png
Good	Good	99,00%	100,00%	good49.png
Good	Good	99,00%	99,00%	good50.png
Bad	Bad	94,00%	94,00%	bad1.png
Bad	Bad	91,00%	98,00%	bad2.png
Bad	Good	65,00%	100,00%	bad3.png
Bad	Bad	97,00%	95,00%	bad4.png
Bad	Bad	93,00%	84,00%	bad5.png
Bad	Bad	96,00%	57,00%	bad6.png
Bad	Bad	40,00%	98,00%	bad7.png
Bad	Bad	91,00%	85,00%	bad8.png
Bad	Bad	49,00%	92,00%	bad9.png
Bad	Bad	26,00%	69,00%	bad10.png
Bad	Bad	64,00%	12,00%	bad11.png
Bad	Bad	24,00%	83,00%	bad12.png
Bad	Bad	92,00%	87,00%	bad13.png
Bad	Bad	92,00%	65,00%	bad14.png
Bad	Bad	98,00%	97,00%	bad15.png

Bad	Bad	96,00%	59,00%	bad16.png
Bad	Bad	92,00%	62,00%	bad17.png
Bad	Bad	86,00%	84,00%	bad18.png
Bad	Bad	98,00%	50,00%	bad19.png
Bad	Bad	96,00%	96,00%	bad20.png
Bad	Bad	76,00%	78,00%	bad21.png
Bad	Bad	77,00%	59,00%	bad22.png
Bad	Good	88,00%	85,00%	bad23.png
Bad	Bad	57,00%	91,00%	bad24.png
Bad	Bad	90,00%	72,00%	bad25.png
Bad	Bad	64,00%	88,00%	bad26.png
Bad	Bad	77,00%	75,00%	bad27.png
Bad	Bad	58,00%	68,00%	bad28.png
Bad	Bad	59,00%	56,00%	bad29.png
Bad	Bad	77,00%	59,00%	bad30.png
Bad	Bad	72,00%	86,00%	bad31.png
Bad	Bad	31,00%	72,00%	bad32.png
Bad	Bad	97,00%	65,00%	bad33.png
Bad	Bad	69,00%	58,00%	bad34.png
Bad	Bad	86,00%	36,00%	bad35.png
Bad	Bad	80,00%	53,00%	bad36.png
Bad	Bad	83,00%	45,00%	bad37.png
Bad	Bad	61,00%	71,00%	bad38.png
Bad	Bad	91,00%	89,00%	bad39.png
Bad	Bad	77,00%	98,00%	bad40.png
Bad	Bad	84,00%	51,00%	bad41.png
Bad	Bad	84,00%	83,00%	bad42.png
Bad	Bad	95,00%	95,00%	bad43.png
Bad	Bad	94,00%	60,00%	bad44.png
Bad	Bad	60,00%	76,00%	bad45.png
Bad	Bad	69,00%	34,00%	bad46.png
Bad	Bad	11,00%	86,00%	bad47.png
Bad	Bad	65,00%	41,00%	bad48.png
Bad	Bad	65,00%	78,00%	bad49.png
Bad	Bad	55,00%	55,00%	bad50.png

APPENDIX I: RESULTS FOR FASTER R-CNN WITH RESNET-101 ONLY AXIS MODEL, $exposure = 21857{,}652\ \mu s$

True Class	Predict Class	Upper axis score	Lower axis score	Image
Good	Good	100,00%	100,00%	good1.png
Good	Good	98,00%	100,00%	good2.png
Good	Good	96,00%	100,00%	good3.png
Good	Good	99,00%	100,00%	good4.png
Good	Good	100,00%	100,00%	good5.png
Good	Good	94,00%	100,00%	good6.png
Good	Good	99,00%	98,00%	good7.png
Good	Good	97,00%	99,00%	good8.png
Good	Good	98,00%	100,00%	good9.png
Good	Good	97,00%	100,00%	good10.png
Good	Good	99,00%	100,00%	good11.png
Good	Good	100,00%	100,00%	good12.png
Good	Good	100,00%	100,00%	good13.png
Good	Good	97,00%	80,00%	good14.png
Good	Good	96,00%	100,00%	good15.png
Good	Good	99,00%	100,00%	good16.png
Good	Good	99,00%	100,00%	good17.png
Good	Good	84,00%	100,00%	good18.png
Good	Good	100,00%	69,00%	good19.png
Good	Good	87,00%	100,00%	good20.png
Good	Good	93,00%	100,00%	good21.png
Good	Good	100,00%	100,00%	good22.png
Good	Good	100,00%	97,00%	good23.png
Good	Good	97,00%	100,00%	good24.png
Good	Good	100,00%	100,00%	good25.png
Good	Good	97,00%	100,00%	good26.png
Good	Good	100,00%	100,00%	good27.png
Good	Good	99,00%	100,00%	good28.png
Good	Good	91,00%	100,00%	good29.png
Good	Good	100,00%	100,00%	good30.png

Good	Good	88,00%	100,00%	good31.png
Good	Good	99,00%	96,00%	good32.png
Good	Good	73,00%	99,00%	good33.png
Good	Good	99,00%	100,00%	good34.png
Good	Bad	37,00%	57,00%	good35.png
Good	Good	90,00%	100,00%	good36.png
Good	Good	91,00%	100,00%	good37.png
Good	Good	39,00%	81,00%	good38.png
Good	Good	100,00%	94,00%	good39.png
Good	Good	82,00%	100,00%	good40.png
Good	Good	89,00%	94,00%	good41.png
Good	Good	60,00%	99,00%	good42.png
Good	Good	100,00%	89,00%	good43.png
Good	Good	96,00%	100,00%	good44.png
Good	Good	96,00%	100,00%	good45.png
Good	Bad	30,00%	50,00%	good46.png
Good	Good	93,00%	99,00%	good47.png
Good	Good	100,00%	100,00%	good48.png
Good	Good	99,00%	96,00%	good49.png
Good	Good	60,00%	70,00%	good50.png
Bad	Bad	97,00%	98,00%	bad1.png
Bad	Bad	98,00%	96,00%	bad2.png
Bad	Good	91,00%	75,00%	bad3.png
Bad	Bad	93,00%	78,00%	bad4.png
Bad	Bad	48,00%	55,00%	bad5.png
Bad	Bad	99,00%	100,00%	bad6.png
Bad	Good	88,00%	88,00%	bad7.png
Bad	Bad	95,00%	97,00%	bad8.png
Bad	Bad	93,00%	64,00%	bad9.png
Bad	Bad	96,00%	29,00%	bad10.png
Bad	Bad	76,00%	45,00%	bad11.png
Bad	Bad	97,00%	99,00%	bad12.png
Bad	Good	71,00%	92,00%	bad13.png
Bad	Bad	98,00%	95,00%	bad14.png
Bad	Bad	70,00%	92,00%	bad15.png

Bad	Bad	98,00%	100,00%	bad16.png
Bad	Bad	93,00%	97,00%	bad17.png
Bad	Bad	98,00%	99,00%	bad18.png
Bad	Good	77,00%	99,00%	bad19.png
Bad	Bad	96,00%	72,00%	bad20.png
Bad	Bad	100,00%	96,00%	bad21.png
Bad	Bad	69,00%	84,00%	bad22.png
Bad	Good	79,00%	100,00%	bad23.png
Bad	Bad	98,00%	53,00%	bad24.png
Bad	Bad	87,00%	57,00%	bad25.png
Bad	Bad	98,00%	91,00%	bad26.png
Bad	Bad	74,00%	99,00%	bad27.png
Bad	Bad	93,00%	80,00%	bad28.png
Bad	Bad	84,00%	99,00%	bad29.png
Bad	Bad	66,00%	90,00%	bad30.png
Bad	Bad	89,00%	61,00%	bad31.png
Bad	Bad	51,00%	88,00%	bad32.png
Bad	Bad	86,00%	94,00%	bad33.png
Bad	Bad	92,00%	55,00%	bad34.png
Bad	Bad	95,00%	62,00%	bad35.png
Bad	Bad	86,00%	82,00%	bad36.png
Bad	Bad	91,00%	98,00%	bad37.png
Bad	Bad	96,00%	94,00%	bad38.png
Bad	Bad	57,00%	50,00%	bad39.png
Bad	Bad	81,00%	89,00%	bad40.png
Bad	Bad	70,00%	65,00%	bad41.png
Bad	Good	85,00%	97,00%	bad42.png
Bad	Bad	45,00%	92,00%	bad43.png
Bad	Bad	87,00%	77,00%	bad44.png
Bad	Bad	99,00%	97,00%	bad45.png
Bad	Bad	84,00%	77,00%	bad46.png
Bad	Bad	99,00%	79,00%	bad47.png
Bad	Bad	95,00%	93,00%	bad48.png
Bad	Bad	74,00%	98,00%	bad49.png
Bad	Bad	73,00%	91,00%	bad50.png

APPENDIX J: RESULTS FOR FASTER R-CNN WITH RESNET-101 ONLY AXIS MODEL, $exposure = 32002{,}609\ \mu s$

True Class	Predict Class	Upper axis score	Lower axis score	Image
Good	Bad	98,00%	93,00%	good1.png
Good	Bad	99,00%	46,00%	good2.png
Good	Good	96,00%	96,00%	good3.png
Good	Bad	97,00%	96,00%	good4.png
Good	Good	99,00%	99,00%	good5.png
Good	Good	96,00%	100,00%	good6.png
Good	Good	78,00%	23,00%	good7.png
Good	Good	96,00%	100,00%	good8.png
Good	Good	100,00%	96,00%	good9.png
Good	Good	99,00%	100,00%	good10.png
Good	Good	98,00%	100,00%	good11.png
Good	Good	97,00%	100,00%	good12.png
Good	Bad	97,00%	64,00%	good13.png
Good	Good	99,00%	16,00%	good14.png
Good	Good	99,00%	99,00%	good15.png
Good	Good	98,00%	100,00%	good16.png
Good	Good	97,00%	99,00%	good17.png
Good	Good	99,00%	99,00%	good18.png
Good	Good	98,00%	98,00%	good19.png
Good	Good	97,00%	100,00%	good20.png
Good	Good	96,00%	99,00%	good21.png
Good	Good	94,00%	16,00%	good22.png
Good	Good	96,00%	95,00%	good23.png
Good	Good	97,00%	100,00%	good24.png
Good	Good	81,00%	96,00%	good25.png
Good	Good	97,00%	100,00%	good26.png
Good	Good	99,00%	98,00%	good27.png
Good	Good	98,00%	99,00%	good28.png
Good	Good	99,00%	90,00%	good29.png
Good	Bad	98,00%	0	good30.png

Good	Bad	99,00%	50,00%	good31.png
Good	Good	99,00%	100,00%	good32.png
Good	Good	99,00%	86,00%	good33.png
Good	Good	100,00%	95,00%	good34.png
Good	Good	98,00%	100,00%	good35.png
Good	Good	98,00%	80,00%	good36.png
Good	Bad	96,00%	83,00%	good37.png
Good	Good	97,00%	32,00%	good38.png
Good	Good	98,00%	77,00%	good39.png
Good	Good	97,00%	100,00%	good40.png
Good	Good	72,00%	97,00%	good41.png
Good	Good	99,00%	84,00%	good42.png
Good	Good	98,00%	79,00%	good43.png
Good	Good	99,00%	100,00%	good44.png
Good	Good	100,00%	97,00%	good45.png
Good	Bad	85,00%	51,00%	good46.png
Good	Good	99,00%	86,00%	good47.png
Good	Good	97,00%	96,00%	good48.png
Good	Bad	94,00%	85,00%	good49.png
Good	Good	99,00%	99,00%	good50.png
Bad	Bad	100,00%	63,00%	bad1.png
Bad	Bad	97,00%	75,00%	bad2.png
Bad	Bad	95,00%	87,00%	bad3.png
Bad	Bad	84,00%	52,00%	bad4.png
Bad	Bad	99,00%	98,00%	bad5.png
Bad	Bad	88,00%	59,00%	bad6.png
Bad	Bad	89,00%	99,00%	bad7.png
Bad	Bad	85,00%	44,00%	bad8.png
Bad	Bad	98,00%	0	bad9.png
Bad	Bad	99,00%	83,00%	bad10.png
Bad	Bad	97,00%	63,00%	bad11.png
Bad	Bad	99,00%	0	bad12.png
Bad	Bad	90,00%	68,00%	bad13.png
Bad	Bad	95,00%	74,00%	bad14.png
Bad	Bad	97,00%	88,00%	bad15.png

Bad	Bad	94,00%	56,00%	bad16.png
Bad	Bad	99,00%	37,00%	bad17.png
Bad	Bad	98,00%	60,00%	bad18.png
Bad	Bad	98,00%	31,00%	bad19.png
Bad	Bad	92,00%	32,00%	bad20.png
Bad	Bad	96,00%	44,00%	bad21.png
Bad	Bad	95,00%	59,00%	bad22.png
Bad	Bad	98,00%	58,00%	bad23.png
Bad	Bad	93,00%	37,00%	bad24.png
Bad	Bad	65,00%	57,00%	bad25.png
Bad	Bad	99,00%	91,00%	bad26.png
Bad	Bad	98,00%	96,00%	bad27.png
Bad	Bad	98,00%	42,00%	bad28.png
Bad	Bad	57,00%	0	bad29.png
Bad	Bad	100,00%	73,00%	bad30.png
Bad	Bad	100,00%	67,00%	bad31.png
Bad	Bad	99,00%	38,00%	bad32.png
Bad	Bad	99,00%	99,00%	bad33.png
Bad	Bad	100,00%	64,00%	bad34.png
Bad	Bad	96,00%	46,00%	bad35.png
Bad	Bad	89,00%	56,00%	bad36.png
Bad	Bad	93,00%	73,00%	bad37.png
Bad	Bad	89,00%	76,00%	bad38.png
Bad	Bad	100,00%	49,00%	bad39.png
Bad	Bad	90,00%	70,00%	bad40.png
Bad	Bad	99,00%	92,00%	bad41.png
Bad	Bad	100,00%	95,00%	bad42.png
Bad	Bad	99,00%	95,00%	bad43.png
Bad	Bad	99,00%	15,00%	bad44.png
Bad	Bad	97,00%	83,00%	bad45.png
Bad	Bad	96,00%	95,00%	bad46.png
Bad	Bad	85,00%	79,00%	bad47.png
Bad	Bad	90,00%	75,00%	bad48.png
Bad	Bad	97,00%	80,00%	bad49.png
Bad	Bad	100,00%	57,00%	bad50.png

APPENDIX K: RESULTS FOR FASTER R-CNN WITH INCEPTION RESNET V2 ONLY AXIS MODEL, $exposure = 10002{,}435\ \mu s$

True Class	Predict Class	Upper axis score	Lower axis score	Image
Good	Good	22,00%	89,00%	good1.png
Good	Good	80,00%	98,00%	good2.png
Good	Bad	29,00%	97,00%	good3.png
Good	Good	86,00%	100,00%	good4.png
Good	Bad	0	94,00%	good5.png
Good	Good	0	98,00%	good6.png
Good	Bad	0	97,00%	good7.png
Good	Good	92,00%	83,00%	good8.png
Good	Good	73,00%	95,00%	good9.png
Good	Good	81,00%	99,00%	good10.png
Good	Bad	68,00%	59,00%	good11.png
Good	Bad	0	97,00%	good12.png
Good	Bad	67,00%	66,00%	good13.png
Good	Bad	24,00%	97,00%	good14.png
Good	Good	39,00%	99,00%	good15.png
Good	Good	56,00%	99,00%	good16.png
Good	Good	54,00%	55,00%	good17.png
Good	Good	85,00%	99,00%	good18.png
Good	Good	85,00%	88,00%	good19.png
Good	Good	81,00%	100,00%	good20.png
Good	Good	93,00%	99,00%	good21.png
Good	Good	95,00%	99,00%	good22.png
Good	Bad	49,00%	97,00%	good23.png
Good	Good	90,00%	99,00%	good24.png
Good	Good	90,00%	100,00%	good25.png
Good	Good	93,00%	99,00%	good26.png
Good	Good	90,00%	99,00%	good27.png
Good	Good	88,00%	99,00%	good28.png
Good	Good	75,00%	97,00%	good29.png
Good	Bad	56,00%	92,00%	good30.png

Good	Good	91,00%	98,00%	good31.png
Good	Good	85,00%	97,00%	good32.png
Good	Bad	18,00%	53,00%	good33.png
Good	Bad	46,00%	97,00%	good34.png
Good	Good	94,00%	99,00%	good35.png
Good	Good	81,00%	99,00%	good36.png
Good	Good	92,00%	100,00%	good37.png
Good	Good	97,00%	99,00%	good38.png
Good	Bad	41,00%	92,00%	good39.png
Good	Good	89,00%	99,00%	good40.png
Good	Good	85,00%	99,00%	good41.png
Good	Bad	66,00%	95,00%	good42.png
Good	Good	94,00%	99,00%	good43.png
Good	Good	90,00%	99,00%	good44.png
Good	Good	88,00%	85,00%	good45.png
Good	Bad	74,00%	95,00%	good46.png
Good	Good	90,00%	99,00%	good47.png
Good	Good	74,00%	99,00%	good48.png
Good	Good	81,00%	99,00%	good49.png
Good	Good	76,00%	91,00%	good50.png
Bad	Bad	57,00%	94,00%	bad1.png
Bad	Bad	44,00%	98,00%	bad2.png
Bad	Bad	0	97,00%	bad3.png
Bad	Bad	37,00%	97,00%	bad4.png
Bad	Bad	30,00%	94,00%	bad5.png
Bad	Bad	60,00%	98,00%	bad6.png
Bad	Bad	0	97,00%	bad7.png
Bad	Bad	40,00%	95,00%	bad8.png
Bad	Bad	16,00%	96,00%	bad9.png
Bad	Bad	72,00%	97,00%	bad10.png
Bad	Bad	11,00%	97,00%	bad11.png
Bad	Bad	0	97,00%	bad12.png
Bad	Bad	60,00%	96,00%	bad13.png
Bad	Bad	48,00%	97,00%	bad14.png
Bad	Bad	32,00%	98,00%	bad15.png

Bad	Bad	52,00%	97,00%	bad16.png
Bad	Bad	53,00%	69,00%	bad17.png
Bad	Bad	70,00%	69,00%	bad18.png
Bad	Bad	90,00%	94,00%	bad19.png
Bad	Good	86,00%	60,00%	bad20.png
Bad	Bad	24,00%	95,00%	bad21.png
Bad	Bad	55,00%	93,00%	bad22.png
Bad	Bad	34,00%	94,00%	bad23.png
Bad	Bad	49,00%	95,00%	bad24.png
Bad	Bad	29,00%	93,00%	bad25.png
Bad	Bad	53,00%	93,00%	bad26.png
Bad	Bad	0	97,00%	bad27.png
Bad	Bad	72,00%	97,00%	bad28.png
Bad	Bad	63,00%	97,00%	bad29.png
Bad	Bad	48,00%	96,00%	bad30.png
Bad	Bad	0	97,00%	bad31.png
Bad	Bad	97,00%	97,00%	bad32.png
Bad	Bad	72,00%	95,00%	bad33.png
Bad	Bad	0	98,00%	bad34.png
Bad	Bad	59,00%	97,00%	bad35.png
Bad	Bad	55,00%	97,00%	bad36.png
Bad	Bad	55,00%	97,00%	bad37.png
Bad	Bad	53,00%	97,00%	bad38.png
Bad	Bad	25,00%	98,00%	bad39.png
Bad	Bad	67,00%	95,00%	bad40.png
Bad	Bad	44,00%	95,00%	bad41.png
Bad	Bad	50,00%	90,00%	bad42.png
Bad	Bad	61,00%	95,00%	bad43.png
Bad	Bad	66,00%	54,00%	bad44.png
Bad	Bad	24,00%	96,00%	bad45.png
Bad	Bad	74,00%	98,00%	bad46.png
Bad	Bad	30,00%	97,00%	bad47.png
Bad	Bad	72,00%	97,00%	bad48.png
Bad	Bad	37,00%	90,00%	bad49.png
Bad	Bad	52,00%	97,00%	bad50.png

APPENDIX L: RESULTS FOR FASTER R-CNN WITH INCEPTION RESNET V2 ONLY AXIS MODEL, $exposure = 21857{,}652\,\mu s$

True Class	Predict Class	Upper axis score	Lower axis score	Image
Good	Bad	78,00%	90,00%	good1.png
Good	Bad	89,00%	95,00%	good2.png
Good	Good	96,00%	99,00%	good3.png
Good	Good	98,00%	99,00%	good4.png
Good	Bad	98,00%	68,00%	good5.png
Good	Good	98,00%	98,00%	good6.png
Good	Good	95,00%	96,00%	good7.png
Good	Good	98,00%	86,00%	good8.png
Good	Good	99,00%	99,00%	good9.png
Good	Good	98,00%	83,00%	good10.png
Good	Bad	97,00%	92,00%	good11.png
Good	Good	99,00%	97,00%	good12.png
Good	Good	99,00%	93,00%	good13.png
Good	Bad	95,00%	93,00%	good14.png
Good	Good	97,00%	78,00%	good15.png
Good	Good	99,00%	95,00%	good16.png
Good	Good	79,00%	80,00%	good17.png
Good	Good	97,00%	98,00%	good18.png
Good	Bad	95,00%	80,00%	good19.png
Good	Good	98,00%	99,00%	good20.png
Good	Good	97,00%	99,00%	good21.png
Good	Bad	95,00%	83,00%	good22.png
Good	Bad	98,00%	92,00%	good23.png
Good	Good	98,00%	99,00%	good24.png
Good	Good	99,00%	98,00%	good25.png
Good	Good	98,00%	93,00%	good26.png
Good	Bad	97,00%	94,00%	good27.png
Good	Good	99,00%	99,00%	good28.png
Good	Bad	79,00%	85,00%	good29.png
Good	Bad	94,00%	79,00%	good30.png

Good	Bad	65,00%	96,00%	good31.png
Good	Good	81,00%	98,00%	good32.png
Good	Bad	73,00%	95,00%	good33.png
Good	Good	90,00%	74,00%	good34.png
Good	Bad	38,00%	94,00%	good35.png
Good	Bad	68,00%	87,00%	good36.png
Good	Bad	94,00%	68,00%	good37.png
Good	Bad	39,00%	84,00%	good38.png
Good	Good	95,00%	92,00%	good39.png
Good	Bad	84,00%	92,00%	good40.png
Good	Good	81,00%	96,00%	good41.png
Good	Bad	57,00%	84,00%	good42.png
Good	Good	95,00%	90,00%	good43.png
Good	Bad	88,00%	86,00%	good44.png
Good	Bad	68,00%	84,00%	good45.png
Good	Bad	52,00%	94,00%	good46.png
Good	Bad	63,00%	91,00%	good47.png
Good	Bad	93,00%	62,00%	good48.png
Good	Good	85,00%	84,00%	good49.png
Good	Bad	36,00%	93,00%	good50.png
Bad	Bad	98,00%	94,00%	bad1.png
Bad	Bad	92,00%	96,00%	bad2.png
Bad	Bad	97,00%	97,00%	bad3.png
Bad	Bad	52,00%	95,00%	bad4.png
Bad	Bad	97,00%	95,00%	bad5.png
Bad	Bad	94,00%	97,00%	bad6.png
Bad	Bad	94,00%	97,00%	bad7.png
Bad	Bad	59,00%	95,00%	bad8.png
Bad	Bad	95,00%	97,00%	bad9.png
Bad	Bad	86,00%	96,00%	bad10.png
Bad	Bad	90,00%	96,00%	bad11.png
Bad	Bad	82,00%	97,00%	bad12.png
Bad	Bad	92,00%	93,00%	bad13.png
Bad	Bad	97,00%	92,00%	bad14.png
Bad	Bad	88,00%	95,00%	bad15.png

Bad	Bad	92,00%	93,00%	bad16.png
Bad	Bad	95,00%	93,00%	bad17.png
Bad	Bad	96,00%	96,00%	bad18.png
Bad	Bad	96,00%	95,00%	bad19.png
Bad	Bad	73,00%	96,00%	bad20.png
Bad	Bad	88,00%	96,00%	bad21.png
Bad	Bad	96,00%	94,00%	bad22.png
Bad	Bad	97,00%	95,00%	bad23.png
Bad	Bad	80,00%	96,00%	bad24.png
Bad	Bad	90,00%	94,00%	bad25.png
Bad	Bad	95,00%	96,00%	bad26.png
Bad	Bad	94,00%	97,00%	bad27.png
Bad	Bad	70,00%	95,00%	bad28.png
Bad	Bad	77,00%	98,00%	bad29.png
Bad	Bad	91,00%	95,00%	bad30.png
Bad	Bad	97,00%	92,00%	bad31.png
Bad	Bad	85,00%	96,00%	bad32.png
Bad	Bad	94,00%	96,00%	bad33.png
Bad	Bad	81,00%	97,00%	bad34.png
Bad	Bad	95,00%	97,00%	bad35.png
Bad	Bad	93,00%	96,00%	bad36.png
Bad	Bad	91,00%	95,00%	bad37.png
Bad	Bad	56,00%	97,00%	bad38.png
Bad	Bad	94,00%	96,00%	bad39.png
Bad	Bad	91,00%	96,00%	bad40.png
Bad	Bad	84,00%	98,00%	bad41.png
Bad	Bad	94,00%	96,00%	bad42.png
Bad	Bad	93,00%	97,00%	bad43.png
Bad	Bad	89,00%	96,00%	bad44.png
Bad	Bad	94,00%	94,00%	bad45.png
Bad	Bad	96,00%	97,00%	bad46.png
Bad	Bad	95,00%	96,00%	bad47.png
Bad	Bad	94,00%	96,00%	bad48.png
Bad	Bad	96,00%	98,00%	bad49.png
Bad	Bad	85,00%	96,00%	bad50.png

APPENDIX M: RESULTS FOR FASTER R-CNN WITH INCEPTION RESNET V2 ONLY AXIS MODEL, $exposure = 32002{,}609\ \mu s$

True Class	Predict Class	Upper axis score	Lower axis score	Image
Good	Bad	99,00%	75,00%	good1.png
Good	Bad	98,00%	89,00%	good2.png
Good	Good	98,00%	99,00%	good3.png
Good	Good	99,00%	98,00%	good4.png
Good	Bad	99,00%	88,00%	good5.png
Good	Good	98,00%	99,00%	good6.png
Good	Bad	95,00%	88,00%	good7.png
Good	Bad	99,00%	65,00%	good8.png
Good	Good	99,00%	67,00%	good9.png
Good	Bad	99,00%	86,00%	good10.png
Good	Good	99,00%	99,00%	good11.png
Good	Bad	99,00%	69,00%	good12.png
Good	Good	99,00%	99,00%	good13.png
Good	Bad	98,00%	79,00%	good14.png
Good	Good	98,00%	99,00%	good15.png
Good	Good	99,00%	99,00%	good16.png
Good	Good	99,00%	99,00%	good17.png
Good	Bad	99,00%	92,00%	good18.png
Good	Good	99,00%	97,00%	good19.png
Good	Good	99,00%	99,00%	good20.png
Good	Good	99,00%	91,00%	good21.png
Good	Bad	98,00%	72,00%	good22.png
Good	Good	99,00%	97,00%	good23.png
Good	Bad	99,00%	85,00%	good24.png
Good	Good	99,00%	76,00%	good25.png
Good	Good	99,00%	97,00%	good26.png
Good	Bad	98,00%	89,00%	good27.png
Good	Good	99,00%	89,00%	good28.png
Good	Good	99,00%	99,00%	good29.png
Good	Bad	98,00%	74,00%	good30.png

Good	Good	98,00%	77,00%	good31.png
Good	Good	99,00%	96,00%	good32.png
Good	Bad	99,00%	69,00%	good33.png
Good	Bad	99,00%	90,00%	good34.png
Good	Bad	99,00%	82,00%	good35.png
Good	Good	99,00%	94,00%	good36.png
Good	Good	99,00%	98,00%	good37.png
Good	Bad	97,00%	88,00%	good38.png
Good	Good	98,00%	85,00%	good39.png
Good	Good	99,00%	82,00%	good40.png
Good	Bad	98,00%	78,00%	good41.png
Good	Bad	99,00%	94,00%	good42.png
Good	Good	99,00%	92,00%	good43.png
Good	Bad	99,00%	69,00%	good44.png
Good	Bad	99,00%	73,00%	good45.png
Good	Bad	97,00%	92,00%	good46.png
Good	Good	99,00%	97,00%	good47.png
Good	Good	97,00%	88,00%	good48.png
Good	Good	99,00%	99,00%	good49.png
Good	Bad	99,00%	69,00%	good50.png
Bad	Bad	93,00%	95,00%	bad1.png
Bad	Bad	94,00%	95,00%	bad2.png
Bad	Bad	97,00%	95,00%	bad3.png
Bad	Bad	91,00%	88,00%	bad4.png
Bad	Bad	94,00%	98,00%	bad5.png
Bad	Bad	60,00%	95,00%	bad6.png
Bad	Bad	94,00%	95,00%	bad7.png
Bad	Bad	89,00%	70,00%	bad8.png
Bad	Bad	75,00%	93,00%	bad9.png
Bad	Bad	85,00%	91,00%	bad10.png
Bad	Bad	80,00%	92,00%	bad11.png
Bad	Bad	82,00%	91,00%	bad12.png
Bad	Bad	95,00%	98,00%	bad13.png
Bad	Bad	98,00%	96,00%	bad14.png
Bad	Bad	68,00%	97,00%	bad15.png

Bad	Bad	90,00%	93,00%	bad16.png
Bad	Bad	94,00%	96,00%	bad17.png
Bad	Bad	93,00%	96,00%	bad18.png
Bad	Bad	97,00%	95,00%	bad19.png
Bad	Bad	69,00%	93,00%	bad20.png
Bad	Bad	97,00%	94,00%	bad21.png
Bad	Bad	93,00%	96,00%	bad22.png
Bad	Bad	96,00%	95,00%	bad23.png
Bad	Bad	94,00%	94,00%	bad24.png
Bad	Bad	93,00%	96,00%	bad25.png
Bad	Bad	97,00%	96,00%	bad26.png
Bad	Bad	87,00%	97,00%	bad27.png
Bad	Bad	93,00%	94,00%	bad28.png
Bad	Bad	83,00%	91,00%	bad29.png
Bad	Bad	87,00%	91,00%	bad30.png
Bad	Bad	80,00%	92,00%	bad31.png
Bad	Bad	80,00%	92,00%	bad32.png
Bad	Bad	94,00%	93,00%	bad33.png
Bad	Bad	92,00%	96,00%	bad34.png
Bad	Bad	96,00%	95,00%	bad35.png
Bad	Bad	84,00%	93,00%	bad36.png
Bad	Bad	96,00%	95,00%	bad37.png
Bad	Bad	91,00%	96,00%	bad38.png
Bad	Bad	95,00%	94,00%	bad39.png
Bad	Bad	87,00%	93,00%	bad40.png
Bad	Bad	91,00%	93,00%	bad41.png
Bad	Bad	90,00%	91,00%	bad42.png
Bad	Bad	94,00%	94,00%	bad43.png
Bad	Bad	90,00%	92,00%	bad44.png
Bad	Bad	94,00%	97,00%	bad45.png
Bad	Bad	97,00%	98,00%	bad46.png
Bad	Bad	84,00%	97,00%	bad47.png
Bad	Bad	91,00%	96,00%	bad48.png
Bad	Bad	96,00%	95,00%	bad49.png
Bad	Bad	92,00%	95,00%	bad50.png

APPENDIX N: RESULTS FOR SSD WITH MOBILENETV2 ONLY HANDLE MODEL, $exposure = 10002{,}435\ \mu s$

True Class	Predict Class	prediction score	Image
Good	Good	70,00%	good1.png
Good	Good	77,00%	good2.png
Good	Good	80,00%	good3.png
Good	Bad	81,00%	good4.png
Good	Good	32,00%	good5.png
Good	Good	41,00%	good6.png
Good	Good	81,00%	good7.png
Good	Good	79,00%	good8.png
Good	Good	77,00%	good9.png
Good	Good	52,00%	good10.png
Good	Good	69,00%	good11.png
Good	Good	50,00%	good12.png
Good	Good	67,00%	good13.png
Good	Good	80,00%	good14.png
Good	Good	31,00%	good15.png
Good	Bad	84,00%	bad1.png
Good	Good	75,00%	bad2.png
Good	Bad	88,00%	bad3.png
Good	Good	55,00%	bad4.png
Good	Good	72,00%	bad5.png
Good	Bad	81,00%	bad6.png
Good	Good	82,00%	bad7.png
Good	Good	53,00%	bad8.png
Good	Good	0	bad9.png
Good	Good	76,00%	bad10.png
Bad	Good	64,00%	bad51.png
Bad	Good	71,00%	bad52.png
Bad	Good	82,00%	bad53.png
Bad	Good	84,00%	bad54.png
Bad	Good	80,00%	bad55.png

Bad	Good	81,00%	bad56.png
Bad	Good	81,00%	bad57.png
Bad	Good	78,00%	bad58.png
Bad	Good	85,00%	bad59.png
Bad	Good	72,00%	bad60.png
Bad	Good	84,00%	bad61.png
Bad	Good	64,00%	bad62.png
Bad	Good	82,00%	bad63.png
Bad	Good	64,00%	bad64.png
Bad	Good	86,00%	bad65.png
Bad	Good	84,00%	bad66.png
Bad	Good	75,00%	bad67.png
Bad	Bad	88,00%	bad68.png
Bad	Good	35,00%	bad69.png
Bad	Good	63,00%	bad70.png
Bad	Bad	87,00%	bad71.png
Bad	Good	88,00%	bad72.png
Bad	Good	73,00%	bad73.png
Bad	Good	64,00%	bad74.png
Bad	Good	45,00%	bad75.png

APPENDIX O: RESULTS FOR SSD WITH MOBILENETV2 ONLY HANDLE MODEL, $exposure = 21857{,}652\ \mu s$

True Class	Predict Class	prediction score	Image
Good	Good	90,00%	good1.png
Good	Good	87,00%	good2.png
Good	Good	96,00%	good3.png
Good	Good	95,00%	good4.png
Good	Good	89,00%	good5.png
Good	Good	76,00%	good6.png
Good	Good	80,00%	good7.png
Good	Good	89,00%	good8.png
Good	Good	94,00%	good9.png
Good	Good	89,00%	good10.png
Good	Good	84,00%	good11.png
Good	Good	95,00%	good12.png
Good	Good	94,00%	good13.png
Good	Good	85,00%	good14.png
Good	Good	91,00%	good15.png
Good	Good	95,00%	bad1.png
Good	Good	95,00%	bad2.png
Good	Good	95,00%	bad3.png
Good	Good	94,00%	bad4.png
Good	Good	95,00%	bad5.png
Good	Good	83,00%	bad6.png
Good	Good	95,00%	bad7.png
Good	Good	80,00%	bad8.png
Good	Good	86,00%	bad9.png
Good	Good	90,00%	bad10.png
Bad	Good	94,00%	bad51.png
Bad	Bad	82,00%	bad52.png
Bad	Good	94,00%	bad53.png
Bad	Good	91,00%	bad54.png
Bad	Good	89,00%	bad55.png

Bad	Good	92,00%	bad56.png
Bad	Good	91,00%	bad57.png
Bad	Good	68,00%	bad58.png
Bad	Good	63,00%	bad59.png
Bad	Good	50,00%	bad60.png
Bad	Good	0	bad61.png
Bad	Good	33,00%	bad62.png
Bad	Good	45,00%	bad63.png
Bad	Good	81,00%	bad64.png
Bad	Good	82,00%	bad65.png
Bad	Good	55,00%	bad66.png
Bad	Good	58,00%	bad67.png
Bad	Good	35,00%	bad68.png
Bad	Bad	67,00%	bad69.png
Bad	Bad	63,00%	bad70.png
Bad	Good	87,00%	bad71.png
Bad	Good	62,00%	bad72.png
Bad	Good	63,00%	bad73.png
Bad	Good	0	bad74.png
Bad	Good	0	bad75.png

APPENDIX P: RESULTS FOR SSD WITH MOBILENETV2 ONLY HANDLE MODEL, $exposure = 32002{,}609\ \mu s$

True Class	Predict Class	prediction score	Image
Good	Good	91,00%	good1.png
Good	Good	81,00%	good2.png
Good	Good	89,00%	good3.png
Good	Good	86,00%	good4.png
Good	Good	93,00%	good5.png
Good	Good	90,00%	good6.png
Good	Good	90,00%	good7.png
Good	Good	82,00%	good8.png
Good	Good	90,00%	good9.png
Good	Good	87,00%	good10.png
Good	Good	96,00%	good11.png
Good	Good	86,00%	good12.png
Good	Good	89,00%	good13.png
Good	Good	89,00%	good14.png
Good	Good	90,00%	good15.png
Good	Good	92,00%	bad1.png
Good	Good	85,00%	bad2.png
Good	Good	98,00%	bad3.png
Good	Good	86,00%	bad4.png
Good	Good	94,00%	bad5.png
Good	Good	91,00%	bad6.png
Good	Good	97,00%	bad7.png
Good	Good	88,00%	bad8.png
Good	Good	88,00%	bad9.png
Good	Good	94,00%	bad10.png
Bad	Good	92,00%	bad51.png
Bad	Good	83,00%	bad52.png
Bad	Bad	89,00%	bad53.png
Bad	Good	77,00%	bad54.png
Bad	Good	87,00%	bad55.png

Bad	Good	86,00%	bad56.png
Bad	Good	94,00%	bad57.png
Bad	Good	82,00%	bad58.png
Bad	Good	81,00%	bad59.png
Bad	Good	84,00%	bad60.png
Bad	Bad	79,00%	bad61.png
Bad	Good	84,00%	bad62.png
Bad	Good	83,00%	bad63.png
Bad	Good	52,00%	bad64.png
Bad	Good	89,00%	bad65.png
Bad	Good	0	bad66.png
Bad	Good	73,00%	bad67.png
Bad	Good	57,00%	bad68.png
Bad	Good	35,00%	bad69.png
Bad	Good	87,00%	bad70.png
Bad	Good	90,00%	bad71.png
Bad	Good	42,00%	bad72.png
Bad	Good	41,00%	bad73.png
Bad	Bad	63,00%	bad74.png
Bad	Good	74,00%	bad75.png

APPENDIX Q: RESULTS FOR FASTER R-CNN WITH RESNET-101 ONLY HANDLE MODEL, $exposure = 10002{,}435\ \mu s$

True Class	Predict Class	prediction score	Image
Good	Good	100,00%	good1.png
Good	Good	100,00%	good2.png
Good	Good	100,00%	good3.png
Good	Good	100,00%	good4.png
Good	Good	100,00%	good5.png
Good	Good	100,00%	good6.png
Good	Good	100,00%	good7.png
Good	Good	100,00%	good8.png
Good	Good	100,00%	good9.png
Good	Good	100,00%	good10.png
Good	Good	100,00%	good11.png
Good	Good	100,00%	good12.png
Good	Good	100,00%	good13.png
Good	Good	100,00%	good14.png
Good	Good	100,00%	good15.png
Good	Good	100,00%	bad1.png
Good	Good	100,00%	bad2.png
Good	Good	100,00%	bad3.png
Good	Good	100,00%	bad4.png
Good	Good	100,00%	bad5.png
Good	Good	100,00%	bad6.png
Good	Good	100,00%	bad7.png
Good	Good	100,00%	bad8.png
Good	Good	100,00%	bad9.png
Good	Good	100,00%	bad10.png
Bad	Bad	98,00%	bad51.png
Bad	Bad	99,00%	bad52.png
Bad	Bad	71,00%	bad53.png
Bad	Bad	73,00%	bad54.png
Bad	Bad	98,00%	bad55.png

Bad	Good	70,00%	bad56.png
Bad	Bad	73,00%	bad57.png
Bad	Bad	99,00%	bad58.png
Bad	Bad	97,00%	bad59.png
Bad	Bad	100,00%	bad60.png
Bad	Bad	97,00%	bad61.png
Bad	Bad	94,00%	bad62.png
Bad	Good	93,00%	bad63.png
Bad	Bad	98,00%	bad64.png
Bad	Bad	99,00%	bad65.png
Bad	Bad	96,00%	bad66.png
Bad	Bad	91,00%	bad67.png
Bad	Bad	100,00%	bad68.png
Bad	Good	86,00%	bad69.png
Bad	Bad	93,00%	bad70.png
Bad	Bad	76,00%	bad71.png
Bad	Bad	97,00%	bad72.png
Bad	Good	81,00%	bad73.png
Bad	Bad	97,00%	bad74.png
Bad	Bad	88,00%	bad75.png

APPENDIX R: RESULTS FOR FASTER R-CNN WITH RESNET-101 ONLY HANDLE MODEL, $exposure = 21857{,}652\ \mu s$

True Class	Predict Class	prediction score	Image
Good	Good	100,00%	good1.png
Good	Good	100,00%	good2.png
Good	Good	100,00%	good3.png
Good	Good	100,00%	good4.png
Good	Good	100,00%	good5.png
Good	Good	100,00%	good6.png
Good	Good	100,00%	good7.png
Good	Good	100,00%	good8.png
Good	Good	100,00%	good9.png
Good	Good	100,00%	good10.png
Good	Good	100,00%	good11.png
Good	Good	100,00%	good12.png
Good	Good	100,00%	good13.png
Good	Good	100,00%	good14.png
Good	Good	100,00%	good15.png
Good	Good	100,00%	bad1.png
Good	Good	100,00%	bad2.png
Good	Good	100,00%	bad3.png
Good	Good	100,00%	bad4.png
Good	Good	100,00%	bad5.png
Good	Good	100,00%	bad6.png
Good	Good	100,00%	bad7.png
Good	Good	100,00%	bad8.png
Good	Good	100,00%	bad9.png
Good	Good	100,00%	bad10.png
Bad	Bad	100,00%	bad51.png
Bad	Bad	100,00%	bad52.png
Bad	Bad	96,00%	bad53.png
Bad	Bad	100,00%	bad54.png
Bad	Bad	100,00%	bad55.png

Bad	Bad	99,00%	bad56.png
Bad	Bad	100,00%	bad57.png
Bad	Bad	99,00%	bad58.png
Bad	Bad	100,00%	bad59.png
Bad	Bad	100,00%	bad60.png
Bad	Bad	99,00%	bad61.png
Bad	Bad	100,00%	bad62.png
Bad	Bad	96,00%	bad63.png
Bad	Bad	87,00%	bad64.png
Bad	Bad	100,00%	bad65.png
Bad	Bad	90,00%	bad66.png
Bad	Bad	100,00%	bad67.png
Bad	Bad	99,00%	bad68.png
Bad	Bad	100,00%	bad69.png
Bad	Bad	100,00%	bad70.png
Bad	Bad	100,00%	bad71.png
Bad	Bad	100,00%	bad72.png
Bad	Bad	100,00%	bad73.png
Bad	Bad	100,00%	bad74.png
Bad	Bad	96,00%	bad75.png

APPENDIX S: RESULTS FOR FASTER R-CNN WITH RESNET-101 ONLY HANDLE MODEL, $exposure = 32002{,}609\ \mu s$

True Class	Predict Class	prediction score	Image
Good	Good	100,00%	good1.png
Good	Good	100,00%	good2.png
Good	Good	100,00%	good3.png
Good	Good	100,00%	good4.png
Good	Good	100,00%	good5.png
Good	Good	100,00%	good6.png
Good	Good	100,00%	good7.png
Good	Good	100,00%	good8.png
Good	Good	100,00%	good9.png
Good	Good	100,00%	good10.png
Good	Good	100,00%	good11.png
Good	Good	100,00%	good12.png
Good	Good	100,00%	good13.png
Good	Good	100,00%	good14.png
Good	Good	100,00%	good15.png
Good	Good	100,00%	bad1.png
Good	Good	100,00%	bad2.png
Good	Good	100,00%	bad3.png
Good	Good	100,00%	bad4.png
Good	Good	100,00%	bad5.png
Good	Good	100,00%	bad6.png
Good	Good	100,00%	bad7.png
Good	Good	100,00%	bad8.png
Good	Good	100,00%	bad9.png
Good	Good	100,00%	bad10.png
Bad	Bad	99,00%	bad51.png
Bad	Good	100,00%	bad52.png
Bad	Bad	100,00%	bad53.png
Bad	Bad	82,00%	bad54.png
Bad	Bad	89,00%	bad55.png

Bad	Bad	100,00%	bad56.png
Bad	Bad	100,00%	bad57.png
Bad	Bad	90,00%	bad58.png
Bad	Good	89,00%	bad59.png
Bad	Good	99,00%	bad60.png
Bad	Bad	100,00%	bad61.png
Bad	Good	100,00%	bad62.png
Bad	Good	87,00%	bad63.png
Bad	Bad	99,00%	bad64.png
Bad	Bad	100,00%	bad65.png
Bad	Bad	97,00%	bad66.png
Bad	Bad	98,00%	bad67.png
Bad	Bad	100,00%	bad68.png
Bad	Bad	100,00%	bad69.png
Bad	Bad	100,00%	bad70.png
Bad	Good	100,00%	bad71.png
Bad	Bad	97,00%	bad72.png
Bad	Good	92,00%	bad73.png
Bad	Bad	95,00%	bad74.png
Bad	Bad	99,00%	bad75.png

APPENDIX T: RESULTS FOR FASTER R-CNN WITH INCEPTION RESNET V2 ONLY HANDLE MODEL, $exposure = 10002{,}435\ \mu s$

True Class	Predict Class	prediction score	Image
Good	Good	99,00%	good1.png
Good	Good	100,00%	good2.png
Good	Good	98,00%	good3.png
Good	Good	99,00%	good4.png
Good	Good	99,00%	good5.png
Good	Good	100,00%	good6.png
Good	Good	99,00%	good7.png
Good	Good	99,00%	good8.png
Good	Good	99,00%	good9.png
Good	Good	100,00%	good10.png
Good	Good	99,00%	good11.png
Good	Good	99,00%	good12.png
Good	Good	99,00%	good13.png
Good	Good	99,00%	good14.png
Good	Good	99,00%	good15.png
Good	Good	99,00%	bad1.png
Good	Good	99,00%	bad2.png
Good	Good	99,00%	bad3.png
Good	Good	99,00%	bad4.png
Good	Good	99,00%	bad5.png
Good	Good	99,00%	bad6.png
Good	Good	99,00%	bad7.png
Good	Good	99,00%	bad8.png
Good	Good	99,00%	bad9.png
Good	Good	99,00%	bad10.png
Bad	Bad	100,00%	bad51.png
Bad	Bad	99,00%	bad52.png
Bad	Bad	99,00%	bad53.png
Bad	Bad	99,00%	bad54.png
Bad	Bad	100,00%	bad55.png

Bad	Bad	99,00%	bad56.png
Bad	Bad	99,00%	bad57.png
Bad	Bad	99,00%	bad58.png
Bad	Bad	99,00%	bad59.png
Bad	Bad	99,00%	bad60.png
Bad	Bad	99,00%	bad61.png
Bad	Bad	99,00%	bad62.png
Bad	Bad	99,00%	bad63.png
Bad	Bad	99,00%	bad64.png
Bad	Bad	100,00%	bad65.png
Bad	Bad	100,00%	bad66.png
Bad	Bad	99,00%	bad67.png
Bad	Bad	100,00%	bad68.png
Bad	Bad	99,00%	bad69.png
Bad	Bad	99,00%	bad70.png
Bad	Bad	98,00%	bad71.png
Bad	Bad	100,00%	bad72.png
Bad	Bad	98,00%	bad73.png
Bad	Bad	99,00%	bad74.png
Bad	Bad	100,00%	bad75.png

APPENDIX U: RESULTS FOR FASTER R-CNN WITH INCEPTION RESNET V2 ONLY HANDLE MODEL, $exposure = 21857{,}652\ \mu s$

True Class	Predict Class	prediction score	Image
Good	Good	100,00%	good1.png
Good	Good	99,00%	good2.png
Good	Good	100,00%	good3.png
Good	Good	100,00%	good4.png
Good	Good	100,00%	good5.png
Good	Good	100,00%	good6.png
Good	Good	99,00%	good7.png
Good	Good	100,00%	good8.png
Good	Good	99,00%	good9.png
Good	Good	100,00%	good10.png
Good	Good	99,00%	good11.png
Good	Good	100,00%	good12.png
Good	Good	100,00%	good13.png
Good	Good	99,00%	good14.png
Good	Good	100,00%	good15.png
Good	Good	100,00%	bad1.png
Good	Good	100,00%	bad2.png
Good	Good	99,00%	bad3.png
Good	Good	100,00%	bad4.png
Good	Good	100,00%	bad5.png
Good	Good	100,00%	bad6.png
Good	Good	100,00%	bad7.png
Good	Good	99,00%	bad8.png
Good	Good	99,00%	bad9.png
Good	Good	99,00%	bad10.png
Bad	Bad	100,00%	bad51.png
Bad	Bad	100,00%	bad52.png
Bad	Bad	100,00%	bad53.png
Bad	Bad	100,00%	bad54.png
Bad	Bad	100,00%	bad55.png

Bad	Bad	100,00%	bad56.png
Bad	Bad	100,00%	bad57.png
Bad	Bad	99,00%	bad58.png
Bad	Bad	100,00%	bad59.png
Bad	Bad	100,00%	bad60.png
Bad	Bad	99,00%	bad61.png
Bad	Bad	100,00%	bad62.png
Bad	Bad	100,00%	bad63.png
Bad	Bad	99,00%	bad64.png
Bad	Bad	100,00%	bad65.png
Bad	Bad	99,00%	bad66.png
Bad	Bad	100,00%	bad67.png
Bad	Bad	99,00%	bad68.png
Bad	Bad	99,00%	bad69.png
Bad	Bad	100,00%	bad70.png
Bad	Bad	100,00%	bad71.png
Bad	Bad	100,00%	bad72.png
Bad	Bad	100,00%	bad73.png
Bad	Bad	99,00%	bad74.png
Bad	Bad	97,00%	bad75.png

APPENDIX V: RESULTS FOR FASTER R-CNN WITH INCEPTION RESNET V2 ONLY HANDLE MODEL, $exposure = 32002{,}609\ \mu s$

True Class	Predict Class	prediction score	Image
Good	Good	99,00%	good1.png
Good	Good	99,00%	good2.png
Good	Good	100,00%	good3.png
Good	Good	100,00%	good4.png
Good	Good	100,00%	good5.png
Good	Good	100,00%	good6.png
Good	Good	100,00%	good7.png
Good	Good	100,00%	good8.png
Good	Good	100,00%	good9.png
Good	Good	100,00%	good10.png
Good	Good	100,00%	good11.png
Good	Good	100,00%	good12.png
Good	Good	100,00%	good13.png
Good	Good	100,00%	good14.png
Good	Good	100,00%	good15.png
Good	Good	100,00%	bad1.png
Good	Good	100,00%	bad2.png
Good	Good	100,00%	bad3.png
Good	Good	100,00%	bad4.png
Good	Good	100,00%	bad5.png
Good	Good	100,00%	bad6.png
Good	Good	100,00%	bad7.png
Good	Good	100,00%	bad8.png
Good	Good	100,00%	bad9.png
Good	Good	100,00%	bad10.png
Bad	Bad	100,00%	bad51.png
Bad	Bad	100,00%	bad52.png
Bad	Bad	100,00%	bad53.png
Bad	Bad	100,00%	bad54.png
Bad	Bad	100,00%	bad55.png

Bad	Bad	100,00%	bad56.png
Bad	Bad	100,00%	bad57.png
Bad	Bad	100,00%	bad58.png
Bad	Bad	100,00%	bad59.png
Bad	Bad	100,00%	bad60.png
Bad	Bad	100,00%	bad61.png
Bad	Bad	99,00%	bad62.png
Bad	Bad	100,00%	bad63.png
Bad	Bad	100,00%	bad64.png
Bad	Bad	100,00%	bad65.png
Bad	Bad	100,00%	bad66.png
Bad	Bad	100,00%	bad67.png
Bad	Bad	100,00%	bad68.png
Bad	Bad	99,00%	bad69.png
Bad	Bad	100,00%	bad70.png
Bad	Bad	98,00%	bad71.png
Bad	Bad	100,00%	bad72.png
Bad	Bad	98,00%	bad73.png
Bad	Bad	100,00%	bad74.png
Bad	Bad	100,00%	bad75.png